Traffic Engineering of MPLS Demands

KEVIN MITCHELL
Agilent Laboratories

The TOAD is a prototype of an offline traffic-engineering system for MPLS being developed within Agilent Labs. It supports demand splitting, with multiple ingresses and egresses. Both edge- and path- based optimization technqiques are supported, making it particularly suitable for environments with multiple traffic classes and a wide range of QoS constraints. The report describes some of the assumptions underpinning the TOAD and a brief overview of its features. The report concludes with some examples of its use and performance.

Key Words and Phrases: MPLS, traffic engineering, optimization, multi-commodity flows

1. INTRODUCTION

The TOAD, a Traffic Optimizer for Aggregated Demands, is a research prototype, developed within Agilent Labs, to explore some of the issues involved in the offline optimization of MPLS tunnels. This paper describes the design of the TOAD, the assumptions underpinning this design, and some preliminary results that have been obtained from the prototype. MPLS[Rosen et al. 2001] fuses the intelligence of routing with the performance of switching, and has rapidly become a key technology for provisioning and managing core networks. We assume the reader is already familiar with label switching technology in general, and MPLS in particular. For those readers requiring a refresher, the book by Davie and Rekhter[Davie and Rekhter 2000] provides a concise overview of the label switching approach. Traffic Engineering (TE) is the process of selecting the paths chosen by data traffic in order to facilitate efficient and reliable network operations, while simultaneously optimizing network resource utilization and traffic performance. In the context of MPLS, traffic engineering involves assigning traffic trunks to label switched paths (LSPs), and the routing of these paths. Quoting Eric Dean from Global One, "MPLS and Traffic Engineering allows for one to spread the traffic and distribute it across the entire network infrastructure like magnetic fields between poles, while also providing the redundancy required for high availability service".

Historically, connectionless data networks have not had to offer high Quality of Service (QoS) guarantees; nor have they needed to support services that require assurances of bandwidth availability. Real-time services, such as voice and video, impose throughput and delay constraints on a network. QoS can be provided on top of an existing network by means of over-provisioning, increasing capacity to ensure that congestion never occurs. MPLS TE allows us to assign traffic from different classes to distinct paths, providing a mechanism to distinguish between these classes, and meet QoS requirements, without needing excessive overprovisioning.

Establishing a label switched path is a relatively expensive operation, both in terms of the signaling traffic required to establish and maintain the path, and the state that needs to be held in all the routers along this path. This suggests that LSPs will not usually be constructed for individual microflows. A more plausible usage model establishes a set of relatively long-lived LSPs, where long in this context may be hours or days. Individual microflows are then tunneled over the most appropriate LSP. Where do the requirements for LSP creation come from in this model? In a VPN setting we might deploy a new set of LSPs as part of the commissioning process for a new corporate customer. In other cases, we might construct traffic matrices between edge routers, and use these to suggest the need for new LSPs. In the context of VoIP, the primary target of the TOAD prototype, the traffic matrices can be constructed by analysing the signaling traffic from the voice gateways[Pollock 2004].

Author's address: K. Mitchell, Agilent Labs Scotland, South Queensferry, Scotland EH30 9TG

The core component of any MPLS system is the label switched path (LSP). This establishes a route across the network, from an ingress to an egress, via a sequence of label switching routers (LSRs). Filters at the ingress control which flows are routed to each LSP. The route taken by an LSP is not constrained to be the same as that followed by best-effort, IGP shortest-hop packets, allowing traffic to be routed away from bottle-necks. There are a number of different mechanisms that can be used to establish these LSPs. The most primitive approach establishes an LSP piece-wise, by configuring label bindings on each of the LSRs along the route. Whilst satisfactory for establishing long-lived LSPs, particularly those whose routes have been determined mechanically, the static approach is not very flexible. This has led to the development of constraint-based routing (CBR) mechanisms. These make use of path establishment protocols, such as RSVP-TE[Awduche et al. 2001] and CR-LDP[Aboul-Magd et al. 2001], to set up paths with reserved bandwidth from ingress to egress.

The input to the CBR process may, at one extreme, just specify the two endpoints of the tunnel, along with a (possibly empty) set of QoS constraints that define the acceptability of any path chosen by CBR. At the other extreme, the constraints may include the exact path to be followed by this LSP, with the CBR step merely being used to reserve resources along this path. Strict routing allows us to control the precise route taken by an LSP, but at the expense of making the solution vulnerable to failures. To address this issue, MPLS allows multiple paths to be specified, with the primary path being used initially, and secondary paths used to support the tunnel in the case of failures. In many cases, it may be preferable to use loose routing, where the path to be followed is only partially specified, allowing CBR to construct solutions that can adapt to changing network conditions.

The CBR approach, at least in its simplest form, uses a "greedy" algorithm. It may route an LSP in a way that blocks future requests where another alternative could have prevented this, or where it may have been better to reject this request for the greater good. For example, consider a set of tunnel requests, or demands, d_1, d_2, ... If we route these demands sequentially then we may find the path used for d_1 prevents d_2 from being placed at all, or it may have to be routed via a suboptimal path. In some cases, there may be an alternative route for d_1 that still satisfies the QoS constraints, but also allows a more optimal routing for d_2. Another disadvantage of CBR is that it typically has to select a path under tight time constraints. Of course this doesn't always have to be the case. When provisioning a new VPN we may need to request the provisioning of multiple LSPs simultaneously, and it may be acceptable to take a few minutes to satisfy this request. But in other cases the request for an LSP may be to satisfy a resource-intensive application, where a response may be required in a small number of seconds. Unfortunately, we are not aware of any mechanism for providing such "timeliness" constraints, or indeed for batching requests.

In some cases, it may be preferable to delegate the routing decisions to a separate route server, for example [Cech 2002]. The Common Open Policy Service (COPS) protocol is a simple query and response protocol that can be used to exchange policy information between a policy server (Policy Decision Point or PDP) and its clients (Policy Enforcement Points or PEPs)[Durham et al. 2000]. In this context COPS can be used as the interface between the router and the route server. An obvious advantage of using a route server is the flexibility it offers. You can change the placement algorithms without altering the routers, and the route server may be able to precompute potential solutions in the idle moments between requests. The server could also compute a backup path after the main LSP has been placed, allowing a faster initial response, although this strategy runs the risk of the primary path failing before the backup path has been configured. The server does not have to wait for OSPF-TE to propagate residual bandwidth changes, and so may be more responsive to such changes. Of course it introduces its own set of problems, such as a single point of failure, and the time taken to/from the server. Distributing the server, either by simple replication, or by IGP area for example, may solve these problems, but at the expense of additional complexity. At present we are not aware of any commercially available solutions based around route servers.

In some situations we may be able to predict the demands between a set of endpoints, allowing offline optimization of their routes. This might be because the traffic is fairly predictable, for example VoIP traffic between sites. Or it might be because some kinds of traffic in the core are sufficiently aggregated that we can view them as consisting of a steady flow with bursts superimposed on top. Other examples may be predictable due to contractual agreements, such as the provisioning of virtual private networks (VPNs). In

these cases, it may be more efficient to calculate the best routes for a collection of LSPs simultaneously, offline.

The offline approach has two main advantages when compared to constraint-based routing. It can place multiple LSPs simultaneously, and so their paths can be chosen to globally minimize the load on the links, for example. Furthermore, because it does not have to run in real-time, it can use more sophisticated optimization algorithms. There are a number of well-established and full-featured applications that perform offline optimization of LSP routes. WANDL's IP/MPLSView[Wandl 2002], CPlane's Traffic Optimizer[CPlane 2003], and OPNET's SP Guru[OPNET 2003] are probably the most successful examples of such tools. In addition to determining the primary path for each LSP, such tools frequently support the optimized routing of backup paths, and link and node protection paths. Offline optimization and capacity planning tools are also useful for "what-if" analysis, allowing the operator to explore the ramifications of different load predictions on the network, or the consequences of various kinds of failure.

An obvious disadvantage of the offline approach is the requirement to identify and predict relatively stable demands in a network that are suitable for optimization. In addition to determining suitable ingress and egress points, and traffic class, we must also construct reasonably accurate bandwidth predictions for these demands. The optimality of the offline approach, when compared to an online strategy, is difficult to quantify, partly because the information available to the routing process varies between CBR implementations. For example, when using a centralized route server the process has global knowledge of what is currently provisioned, and may also have some knowledge of the current loads on individual routers. As this load depends on best-effort traffic demands, a factor not visible to the offline process, in some cases the CBR approach may have more information at its disposal than the offline approach.

From the previous discussion it is clear that each approach has its advantages and disadvantages. Hybrid solutions attempt to combine the best of both approaches. We can distinguish three main forms of hybrid; *spatial*, *temporal*, and CBR approaches guided by offline *pre-computation*.

Perhaps the simplest approach is to exploit the access/core distinction that is common in many networks; we call this the *spatial approach*. We provision a relatively static set of LSPs across the core of the network, and use these to tunnel the more ephemeral LSPs created by the access layer. Offline tools can then be used to engineer the core LSPs, and constraint-based routing to provision the access-level LSPs. The core network will typically be overprovisioned, with plenty of space capacity remaining to support best-effort traffic for example. We can therefore allow some changes to the core configuration, such as modest changes to reserved bandwidth, without needing to reengineer the core. Given the highly aggregated nature of the flows, we may only need to rerun the offline tool once every few days. Furthermore, the need to provision a new LSP across the core, between runs of the offline optimizer, may be a sufficiently rare occurrence that routing it using CBR may be adequate until the next run of the tool. At the access level, the LSPs may be so ephemeral that it becomes pointless placing them using an offline tool. Furthermore, in many cases the routing process may be fairly trivial, as most of the path will consist of a tunnel through the core. In these cases, we merely have to decide on the best tunnel to use, rather than worrying about the individual hops. If our network, and LSP usage patterns, fit into this model then this may be the simplest way of marrying together the offline and online approaches. Using such a technique is predominently a procedural issue; the hybrid introduces no new technical issues that aren't already present in the individual offline and online components.

The *temporal approach* does not make core/access distinctions. Instead, it optimizes those demands it currently knows about simultaneously, offline, and then routes subsequent demands online, using constraint-based routing. Obviously if most of the demandss can be predicted, and routed statically, then such a hybrid will behave similarly to a purely offline tool. If most demands arrive between runs of the offline tool then the system will behave more like a CBR system. But the nature of the requests also affects how such a hybrid performs. A new high-priority request may not be able to be fulfilled without preempting existing lower-priority LSPs. The preempted LSPs will then need to find an alternative route to the egress, or fail. If a preempted LSP has been provisioned as part of the offline process then it may be strictly routed all the way to the egress. This can make it difficult to reroute it when prempted. It may have a backup tunnel, but this is designed to be used in the case of link or node failure. Preemption may not be considered by

some offline tool if they do not consider the possibility of future CBR LSP requests. Some approaches, e.g. [Iovanna et al. 2003], avoid such complexities by only provisioning the highest priority LSPs in the offline phase.

The *pre-computation* approach uses online constraint-based routing to provision every LSP. But it attempts to compute offline information that can be used to speed up, or improve this process. Examples of this approach include [Orda and Sprintson 2000], [Cui et al. 2003] and [Suri et al. 2001]. Approaches based on pre-computation attempt to either speed up the routing process, or to exploit information not necessarily available to individual routers, such as traffic profile details.

2. LIMITATIONS OF EXISTING APPROACHES

Inputs to offline tools typically consist of requests to provision a set of LSPs with particular characteristics. Where do these LSP demands come from? In some cases, they can be generated from traffic matrices. Consider a demand for 50 Mb/s of IP traffic between an ingress and egress point. Such a demand might consist of a single flow, for example as part of a high-performance grid computing environment. But in other situations, such as VoIP, the demand might be generated by the aggregation of a large number of relatively small flows. Offline optimizers usually map a demand to a single LSP, to avoid introducing undesirable packet reordering. Whilst such a restriction makes sense in the context of the single large microflow example, it is less clear it is necessary in the VoIP case. Although each of the individual microflows should follow a single path, and therefore be assigned to just one LSP, if two calls were assigned to different LSPs then no harm would occur. Furthermore, we may be able to make better use of the network by provisioning multiple smaller LSPs than one large one.

This analysis suggests that we might be better off provisioning ten 5 Mb/s LSPs between the ingress and egress, rather than a single 50 Mb/s LSP. Of course we have a trade-off here. A large number of LSPs, each with a smaller bandwidth request, may be easier to route, easier to protect, and make better usage of the overall network resources. But there will also be more signaling overhead. What is the best way of splitting an LSP to optimize network utilization, and how can we distribute the microflows to the resulting LSPs in an efficient fashion? Rather than splitting the demand prior to the optimization stage, it may be preferable to express initial demands in terms of bandwidth, and characteristics of the aggregated traffic making up this demand. We then let the system determine which LSPs to provision to make up this demand, and the mechanism necessary to map microflows to the appropriate LSP.

The previous discussion has considered the scenario where a large demand is best split into multiple smaller demands. But the opposite problem may also occur, particularly in the context of VPNs. In a VPN environment you may assign traffic to different LSPs, to keep them logically separate, even though they follow the same route across an MPLS cloud. Moreover, if every small customer is assigned their own VPN then many of these LSPs will have small bandwidth requirements. The obvious strategy here may be to tunnel these LSPs across another LSP. But tunneling all LSPs that share a common route, and similar traffic class, over a single LSP may yield a very large LSP. For the same reasons as discussed earlier, it may be preferable to tunnel them over more than one LSP. And if so, what is the best tunnel size for a particular topology? These issues are frequently neglected by offline tools.

Each LSP has a fixed egress point. Backup paths are also constrained to have the same egress as the path they are protecting. In some cases, this is a natural restriction. For example, an LSP might be constructed between two sites belonging to the same customer. In such a case there will be a unique ingress and a unique egress. But consider the example where an LSP is provisioned across an MPLS cloud to a border gateway router, providing access to a portion of the external Internet. There may be many such routers that could provide this forwarding service. However, the IGP will typically just select one of them at any particular point in time. Constructing an LSP, and secondary path, to this single egress may be restrictive in a number of ways. If the gateway fails then the LSP itself will also break. If there are other gateways that could forward this traffic it may be desirable to establish a backup path with a different egress point. Furthermore, even without failures, we may reach a situation where the egress point becomes overloaded, or where links become saturated because of the load imposed by the LSP. In such cases, we may find that other egress points, that could potentially carry some of this load, are underutilized. This suggests that it

may be attractive to allow demands with sets of egresses in their specification, as long as we can ensure that each microflow contained within such a demand only uses one of these egresses. Such an idea is not entirely new. The work on NetScope[Feldmann et al. 1999; Feldmann et al. 2000] applied similar arguments to the construction of traffic matrices, but it's ramifications in the context of MPLS have not been explored as far as we are aware.

There is some evidence to suggest that routing instability, due to link and router failure, may have a significant impact on VoIP quality.[Boutremans et al.] The preprovisioning of secondary routes, fast rerouting, and other path protection mechanisms, are one of the main advantages of using MPLS for supporting VoIP services. But constraining such paths to always have the same egress point may result in single points of failure, and/or secondary paths taking very non-optimal routes. This further motivates the study of more flexible LSP specifications.

2.1 The LSP assignment problem

One of the main difficulties with mapping demands to multiple LSPs is the need to construct filters at the ingress router. These are required to map microflows associated with the demand into the LSPs supporting the demand, in the correct proportions dictated by the bandwidth reservations for these LSPs, and without splitting the individual flows. A demand may also be built from a set of LSPs constructed to support one or more VPNs. Let's call these "VPN LSPs" to distinguish them from the "TE LSPs" established to support a demand. A collection of VPN LSPs may be tunneled across a TE LSP, and so filters must also be constructed to perform this association in the correct proportions. We call the mapping of microflows and VPN LSPs to a set of TE LSPs the *LSP assignment problem*. If a demand is made up of many different sizes of microflow, and these are connectionless in nature, then there may be no good mechanism for splitting such a demand without splitting individual microflows. This should be avoided where possible as it increases the risk of packet reordering.

If the traffic making up the demand has any obvious substructure then this could be exploited to partition the traffic into two or more subclasses, and each of these could be mapped to a separate LSP and then routed by a tool like WANDL. In this case individual traffic demands drive the construction of the LSPs, and the effect this decision has on overall network utilization is only considered as an afterthought. In contrast, we propose driving the decomposition process from the global optimization of demands. The challenge in this case is to efficiently map microflows to the resulting set of LSPs. So what kinds of demand might be amenable to such splitting?

In our first scenario consider the case where the demand consists of a large number of connection-oriented microflows, each requiring a similar bandwidth reservation. VoIP streams are a good example of what we have in mind here. At one level it should be fairly easy to map calls to LSPs. It's clear when the flow starts, and the size of the flow is bounded. But the ingress point for the LSP may not be aware of such details, and establishing a separate filter for each call may overload an edge router. As MPLS technology moves towards the access network this becomes less of an issue, as voice gateways and MPLS edge routers become colocated.

In the second scenario consider the case where VPNs are deployed extensively, with many customers being assigned their own VPN LSPs. In such a situation there may be many VPN LSPs with similar QoS requirements traversing paths between a particular ingress and egress. A single TE LSP between the ingress and egress could be established, with a bandwidth reservation equal to the sum of the individual reservations. The original LSPs could then use this LSP as a tunnel. This would potentially reduce the signaling overhead, and conserve label space through the core. But the resulting LSP may require a large bandwidth reservation. If we view this as our demand then there are clearly many other ways of aggregating and tunneling the individual VPN LSPs into/across the TE LSPs constructed for this purpose. We could run a multi-commodity flow (MCF) algorithm to determine a globally optimum number of TE LSPs, and their bandwidth requirements, and then construct an embedding from the VPN LSPs to the TE LSPs. In this scenario it should be relatively simple to create filters to tunnel the VPN LSPs into the appropriate TE LSPs. There will be far fewer of them than individual microflows. Moreover we just need to map a label/ingress interface into the appropriate "label-push" action. In this, and the other scenarios, we assume

that there will be some flexibility in how we choose the bandwidth of the TE LSPs. We may run a MCF algorithm to get an idea of how many LSPs we should construct, and their bandwidth requirements. But we may then adjust these figures by a small amount to make it easier to host the required flows/VPN LSPs. We assume that small changes will not move the solution too far away from the "optimum". We also assume that there will be excess residual capacity on most links, for example to support best-effort traffic and subsequent CBR LSP demands, allowing us to make minor adjustments to bandwidth reservations without saturating a link.

For the final scenario consider the situation where a router has two parallel links to a peer. Routers will sometimes spread the load over these links using hashing to minimize packet reordering. A hash value is computed for each packet header, and this is used to determine which of the links the packet should be forwarded over. The hash function is designed so that each packet in a microflow will hash to the same value. In the extreme case, where all the packets were for a single flow, then one of the links would never be used. However, given a mix of flows, and a carefully chosen hashing function, we can spread the load fairly evenly over the two links. A similar approach could be applied to the LSP assignment problem. Of course this hashing process will not be perfect, and we will find that the distribution of traffic to the individual LSPs will not match the distribution required by their reservations. What can we do about this? Two strategies spring to mind. First, we can dynamically modify the mapping from hash value to LSP in an attempt to move the observed distributions in the required direction. If this fails, and an individual LSP reaches its full capacity, we could use an approach similar to that advocated by [Wang et al. 2002] The idea would be to start assigning excess traffic to one of the other LSPs associated with the demand. This would obviously increase the probability of packet reordering, but to what extent? In a heavily aggregated flow the packets associated with an individual microflow may be spaced sufficiently far apart that they would be unlikly to be reordered as a result of such a mechanism. This scenario would obviously require changes to edge routers, just as techniques like RED and WRED do[Cisco 2002]. But it suggests that, at least in principle, MPLS routers could split demands across multiple LSPs without introducing excessive packet reordering.

The current TOAD prototype focuses on the first of these scenarios, assuming the voice gateways are also MPLS edge routers, e.g. [RAD 2004].

3. DEMANDS

In the context of this report we view a *demand* as a request for a guaranteed amount of bandwidth between an ingress and an egress node, with an associated QoS requirement or constraint. Demands arise from a variety of sources. A request to provision a high-bandwidth video link could be viewed as a demand. Such demands arrive incrementally and, in the context of an MPLS network, would be mapped to label switched paths using an online algorithm such as CSPF with RSVP-TE. Predictions based on an analysis of traffic matrices can yield another source of demands. In this case we typically generate a large collection of demands that will exist for the duration of the prediction period. We could provision each of these demands in turn, using CSPF. But we may be able to improve network utilization by simultaneously optimizing the LSP routes for all these demands. One of the primary goals of the TOAD prototype is to perform such optimizations.

3.1 Demand aggregation

There could be many demands associated with each ingress/egress pair. In a VPN setting individual corporate customers may each generate a set of demands between these endpoints. Furthermore, a demand has a QoS constraint attached to it, recording aspects such as the maximum hop limit, delay, affinity groups, and cost. So even for a single customer there may be multiple demands associated with an ingress/egress pair, corresponding to different service classes and traffic characteristics. The optimizer will associate one or more paths to each demand, and each of these paths will, in turn, be hosted by a single LSP. This is clearly worrying from a scaling viewpoint, as each LSP has a small, but non-trivial, signaling and state overhead in each router along the path.

Allowing small demands to be split over multiple paths increases the risk that we will be unable to assign the constituent flows to the LSPs in the correct proportions without requiring bandwidth adjustments, or

splitting microflows. For these reasons we assume that demands are aggregated, where possible, prior to processing by the optimizer. Demands for different customers may be merged, from the perspective of the optimizer, if they have the same ingress, egress, and QoS constraints. But what about demands that share the same ingress and egress, but have different traffic classes/QoS constraints? If they only differ superficially then we might choose to merge such demands, giving the combined demand the stricter of the QoS settings. Here we are trading off optimality for scalability.

The MPLS specification includes an EXP field within the shim header, allowing two flows traversing the same LSP to be treated differently at each hop if the headers have different EXP bits. This feature could be exploited to allow demands to be merged that have very different QoS constraints, using the EXP bits to provide the service differentiation between the two demands. But a better strategy might be to keep the demands separate in such cases. If the demands are then assigned a common path by the optimizer we could provision a single LSP, using the EXP bits to provide the differentiation. This would reduce the number of provisioned LSPs whilst not unduly constraining the optimizer. At present, the TOAD implementation does not exploit the EXP bits.

3.2 Multiple ingresses and egresses

An LSP has a single exit point, at the egress. But what about demands? Consider the problem of building a traffic matrix between VoIP gateways. These matrices would form the basis of our demands in the VoIP case. A call arriving at an ingress gateway could potentially be forwarded to its final destination by a number of different egress gateways. In some cases one of these would be the preferred gateway to use, whilst in other cases two or more gateways might be equally acceptable. Discarding this valuable information when building the traffic matrices seems wrong[Feldmann et al. 1999; Feldmann et al. 2000]. This suggests that our traffic demands should also support the possibility of multiple egresses, with some indication of preference between them[Pollock 2004]. The result of the optimization process would be a set of paths for each demand. Each path would have a single egress as its endpoint, to enable it to be implemented by a single LSP. However, different paths for the same demand could be allowed to have different termination points. Where one egress is to be preferred over another then we would expect the paths to use the preferred egress. However, using another egress might reduce network utilization on some other aspect of the optimization problem we are also interested in. The emphasis to place on egress choice, compared to the other cost metrics, is specified by the user when the optimization criteria is chosen.

One approach we could adopt would be to extend the notion of demand to allow a set of egresses to be specified explicitly. In some cases we might even require multiple ingresses, for example where demands across a large network are transformed into demands across a smaller core network[Mitchell 2004b]. One disadvantage of such an approach is that it can be difficult to specify egress preferences in a way that interacts predictably with our other optimization goals. To address this issue we follow a different path to supporting multiple endpoints. We allow *virtual nodes* to be added to the topology, along with virtual links that connect these new nodes to the non-virtual ones. Consider the situation where we wish to define a demand from ingress i to two egresses, e_1 and e_2. We construct a new virtual node, $\{e_1, e_2\}$, and add two unidirectional links, one from e_1 to $\{e_1, e_2\}$ and the other from e_2 to the new node. This situation is illustrated in Figure 1.

The costs associated with traversing the virtual links allow us to express preferences between using e_1 and e_2. The costs may be completely artificial, just expressing administrative preferences between the two gateways. Or they may be an approximation of the cost of forwarding the traffic to the final destination using each of these gateways. The TOAD optimizer treats virtual nodes like any other nodes. At the end of the optimization process there will be one or more paths selected to satisfy this demand. For each such path it will either pass through e_1 or e_2. When provisioning the LSPs for these paths we use e_1 or e_2 as the egress, as appropriate; the presence of $\{e_1, e_2\}$ is not visible in the provisioned solution. We also ensure that traversing a virtual link does not increase the hop count, as otherwise this would rule out paths that were perfectly acceptable in the provisioned solution. Demands requiring multiple ingresses can be handled using the same mechanism. The challenges of route optimization in the presence of multiple egresses is discussed further in [Mitchell 2004a].

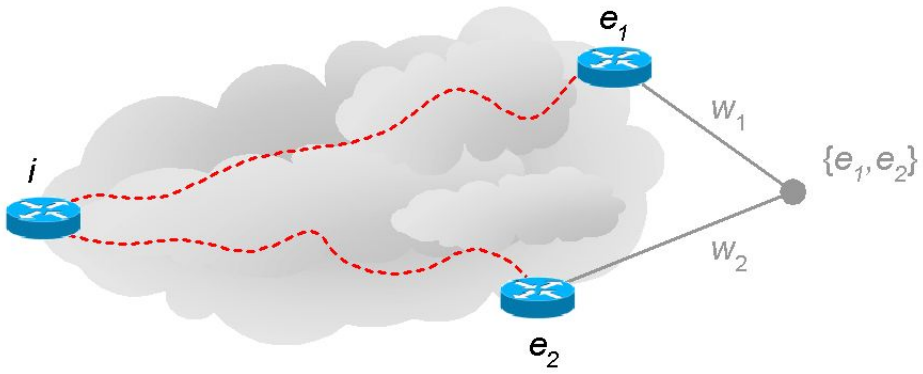

Fig. 1. Representing multiple egresses using virtual nodes and links

The MPLS specification allows secondary or backup paths to be provisioned along with the primary path. In the case of multiple egresses it might be natural to allow the egress of a backup path to be different to the primary path egress. The current TOAD implementation does not address the backup path problem.

3.3 Bidirectional demands

MPLS LSPs are unidirectional. But many demands are naturally bidirectional. In some cases the bandwidth required in the two directions may be different. In other cases, such as aggregated VoIP traffic, both directions can be viewed as requiring the same bandwidth. Consider a demand from I to E with a bandwidth requirement of 100 Mbps. If this denotes a bidirectional demand then there is also an implicit demand of 100 Mbps from E to I to support this traffic.

The situation becomes more complex when bidirectional demands with multiple egresses are required. Consider the example in Figure 1, where the demand from i to $\{e_1, e_2\}$ is now bidirectional, with a bandwidth requirement of 100 Mbps. Converting this to a unidirectional demand, and introducing a second unidirectional demand of 100 Mbps from $\{e_1, e_2\}$ to i, will not always give us the solution we require, depending on the nature of the demand. To see why, consider a solution where all the demand is routed via e_1 in the forward direction and via e_2 in the reverse direction. This would result in one LSP being provisioned from i to e_1 and a second one from e_2 to i. If this demand represents an aggregated collection of VoIP calls, with e_1 and e_2 representing voice gateways, then this is clearly an invalid solution. Each call must leave the network via a single gateway; having the return path enter the network though a different gateway is not allowed. This problem, and a solution, are discussed in more detail in [Mitchell 2004a].

In summary, the TOAD prototype assumes demands can be split across multiple LSPs, and also supports demands with multiple egresses. This positions the prototype in a unique position within the TE optimization space. Other features, such as scripting support, and a methodology for hierarchically decomposing demands [Mitchell 2004b], add further novelty to this work. Although not currently supported, it would be relatively easy to handle demands that should not be split, simply by adding additional constraints to the optimization model.

3.4 Topology and Demand files

To optimize the routing of a set of demands we require access to the network topology. The demands have bottleneck constraints such as bandwidth, and additive constraints such as maximum delay and cost. The links in the network topology must be annotated with attributes to support these metrics. The prototype uses an XML file, backed by an XML Schema, to describe the network topology. A simple example of such a file is shown below.

```
<network
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
```

```
    xsi:noNamespaceSchemaLocation='Network.xsd'>
  <node id='0' name='San Diego' AS='-1' type='RT_NODE' .../>
  <node id='1' name='Palo Alto' AS='-1' type='RT_NODE' .../>
  ...

  <edge delay='2.5688' bw='155' ...>
    <node id='0'/>
    <node id='1'/>
  </edge>

  <edge delay='0.0202' bw='155' ...>
    <node id='1'/>
    <node id='2'/>
  </edge>
  ...
</network>
```

To produce the topology files we currently use a modified version of the IGP Detective[Lehane 2002]. A custom client registers with the agent for a period of time, building up a maximal view of the network elements. At the end of this period it dumps out a topology file that can be loaded into the TOAD. We use the OSPF-TE extensions[Katz et al. 2003] to find the link attributes. Unfortunately, these are often set manually by the operators, and so cannot always be trusted. Furthermore, for large networks spanning multiple autonomous systems we may need to stitch together multiple small topologies, using BGP connectivity information, to provide a complete network topology. At present this step has to be performed manually. Other alternatives exist for collecting this data, including SNMP, and Cisco's CDP. In a real deployment of the optimizer we may be able to retrieve topology information from the OSS system as well.

The demand files define a set of traffic classes, and a set of demands built using these classes. The traffic classes define a set of QoS constraints, currently consisting of a hop limit, maximum delay, and cost. They also have a priority which is used to stratify the optimization task; we first attempt to optimize all the demands whose associated traffic classes have the highest priority, and then repeat the process with the rest of the demands on the residual network. The demand files are written using XML, backed by a Demands schema. An example demand file is presented below.

```
<demands
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='Demands.xsd'>

  <class name='premium-plus' id='0' priority='1' hops='6' delay='1000' cost='20.2'/>
  <class name='premium' id='1' priority='0' hops='6' delay='1000' cost='20.2'/>
  <class name='best-effort' id='2' priority='0' hops='4' delay='3000' cost='40'/>
  <class name='best-effort+' id='3' priority='0' hops='4' delay='3000' cost='40'/>

  <demand tc='1' bw ='10' ...>
    <ingress name='San Diego'/>
    <egress  name='Palo Alto'/>
  </demand>

  <demand tc='2' bw ='5' ...>
    <ingress name='Palo Alto'/>
    <egress  name='Argonne'/>
  </demand>

  ...
</demands>
```

To support multiple ingresses and egresses, we allow additional nodes and edges to be defined within the demand file. These will be treated as virtual nodes and edges. As such, they are constrained so that a virtual edge can only connect a virtual node to a non-virtual node. In the following example we contruct a virtual node to represent a choice between `N1` and `N2`.

```
<demands ...>
  ...
  <node name='N1+N2'/>
  <edge from='N1' to='N1+N2' delay='1' bw='10'/>
  <edge from='N2' to='N1+N2' delay='2' bw='10'/>

  <edge from='N1+N2' to='N1' delay='1' bw='10'/>
  <edge from='N1+N2' to='N2' delay='2' bw='10'/>

  <demand tc='0' bw ='10'>
    <ingress name='N0'/>
    <egress  name='N1+N2'/>
  </demand>

  <demand tc='0' bw ='10'>
    <ingress name='N1+N2'/>
    <egress  name='N0'/>
  </demand>
</demands>
```

4. FLOWS

Consider the problem of optimizing the placement of a set of demands across the network. There are two equivalent ways of representing any solution to this problem from the viewpoint of a particular demand, d. For each arc in the graph we can record how much of the demand should flow over this edge. In an "optimal" solution we would expect this value to be zero for most arcs. We refer to this approach as an *edge-based* solution to the demands problem. But the flow decomposition theorem ([Ahuja et al. 1993]) tells us that any collection of nonnegative arc flows can also be represented as a set of path and cycle flows, though not necessarily uniquely. Therefore we can also describe any solution by a set of paths, cycles, and the amount of flow traversing each of these. Given that we are typically trying to minimize a cost function such as routing cost, or delay, we would not expect cycles in our optimal solutions, and so we can restrict our solutions to just those containing simple paths. An alternative to the edge-based approach is to explicitly enumerate a set of paths between ingress and egress, and then calculate how much of the demand should flow over each of these whilst optimizing some objective function. We refer to this approach as the *path-based* solution to the demands problem.

Both encodings could be used to model the problem, and there are advantages and disadvantages to each approach. We consider each one in turn.

4.1 Edge-based modeling of demands

The obvious advantage of an edge-based encoding is that you do not need to determine, in advance, the set of paths that are likely to be used in an optimal solution. The optimizer is free to assign the demand to any collection of edges that connect the ingress to the egress, subject of course to some flow conservation laws. But there are some disadvantages to this approach, particularly where the demands have conservative QoS constraints attached to them. For each edge, and each demand or (ingress,egress,QoS class) triple, we need a decision variable. Even for a relatively small but densely connected network of a hundred nodes and ten traffic classes we may require hundreds of millions of decision variables. Fortunately these variables will tend to be very sparse, i.e. for any demand most of these decision variables will be 0.

A more serious problem arises when the demands have QoS constraints. Given an edge-based solution to the problem we will need to convert this into a set of paths in order to map these to a set of LSPs. A demand

will often have additive QoS constraints associated with it, such as maximum hop length or delay. It can be extremely difficult to represent some of these constraints in the form of edge constraints and objective functions. Some of these issues are discussed in [Mitchell 2004a]. A simplistic approach might be to ignore such constraints during the optimization process. We then convert the edge solution to paths, and check that each of these paths satisfies the QoS constraints for the demand. If the QoS constraints are not too onerous then we may be fortunate and find all the paths satisfy these constraints. f they don't we may need to discard the unacceptable paths and hope there is still enough residual link capacity to route the demand along a path that does satisfy the constraints. Of course, the more demands that have to be handled in this way, the further we move away from the globally optimum solution. Where the QoS constraints are very strict we may find that the majority of the paths found by the edge-based approach are invalid. In these cases a path-based approach to the problem may be preferable.

4.2 Path-based modeling of demands

In a path-based model we start by computing a set of candidate paths, paths that are likely to form good routes between ingress and egress, whilst respecting the QoS requirements of the demand. The optimizer then allocates the demand to these paths in a globally optimum fashion. In this approach we need a decision variable for each (demand,path) tuple. This may require more or less decision variables than the edge-based approach, depending on how many candidate paths we generate for each demand. The main advantage of such an approach is the ability to select only those paths that could satisfy the QoS constraints of the demand. Unfortunately the optimality of the solution is only as good as the selection of candidate paths. Choose too few and you may miss a better solution. Choose too many and the problem may become intractable.

In a situation where there are few or no constraints on acceptable paths then an edge-based formulation seems preferable. Routing a demand representing best-effort traffic is a good example of where an edge-based formulation is suitable. For demands with strict QoS constraints, such as VoIP traffic, than a path-based approach is often preferable. Some demands could be handled by either approach, and detailed timings need to be performed to ascertain the best approach. An optimization suite needs to support both approaches, choosing the best technique to use for each demand.

5. CANDIDATE PATHS

The path-based model requires a set of candidate paths for each demand. These paths must be chosen to satisfy the QoS requirements associated with the demand. There may be many potential paths between ingress and egress, particularly when the QoS requirements are quite loose, or the network connectivity is very dense. We could select all possible paths that satisfied the QoS constraints, but this would obviously not scale well as the network size increased. At the other extreme, if we select too few paths then we run the risk of not being able to satisfy the demand. Furthermore, even if the demand is satisfied, it may be at the expense of a non-optimal route for one or more of the other demands. As we don't know, in advance, which paths would be chosen in an optimal solution, choosing a set of paths is largely guesswork. The prototype currently uses a variant of the A*Prune algorithm[Liu and Ramakrishnan 2001] to generate a set of candidate paths.

The A*Prune algorithm combines the well-known A*-search with pruning. As paths are expanded out from the root node we use the shortest projected path length to prune paths that cannot lead to a successful solution. Furthermore, the candidate paths are ordered such that the path with the shortest projected length is selected and expanded first. We terminate our expansion procedure once we have found enough acceptable paths, or there are no candidate paths left.

How many candidate paths do we need to generate for each demand? We start with a user-settable limit N, typically < 10. Having found N paths, the total available bandwidth for these paths may still be less than the bandwidth required for the demand. In such cases the implementation attempts to find additional paths until either there is sufficient bandwidth, or too much time has elapsed. Note that this approach is still not guaranteed to succeed. Other demands may also need to be routed over these links, consuming some of this bandwidth. A better approach might be to keep going until the bandwidth supported by the

paths exceeds the required bandwidth by a user-specified factor.

The heuristic just described allows the algorithm to return more than N paths in some cases. However, there are also occasions where we would like to return less than N paths. In some networks searching for N paths that satisfy the QoS constraints can be very time-consuming. Ideally, we would like to find all N paths. However, suppose we reach the state where we have found N' paths, where $N' < N$. If these paths have sufficient bandwidth to satisfy the demand, and we have already spent a lot of time finding them, then it may make sense to terminate the algorithm prematurely. The A*Prune algorithm is iterative in nature, and so we can control the duration by a bound on the number of iterations. Initially, this bound is set to a large number. Once we have found sufficient paths to potentially satisfy the demand we start reducing the bound. Each additional path found reduces the bound further, until we either find all N paths or the number of iterations reaches the current bound.

The algorithm generates the paths ordered by "weight". Where a single attribute is being considered, such as hop count, then this results in the paths being generated ordered by path length. Similarly, if we were considering path cost then the paths would be ordered by cost. Where multiple attributes are being considered simultaneously then the algorithm needs some way of translating the individual metrics into an aggregated weight for ordering purposes. The implementation is parameterized on a function that performs this task. Ideally the weighting given to each of the additive attributes should be under the control of the user, allowing greater emphasis to be placed on path cost for example. At present the weightings are fixed, with the paths generated in increasing path length. Note that this function only controls the order in which paths are generated. If we asked for all paths then we would get the same set, irrespective of the weighting function, and all paths in this set would satisfy the QoS requirements. However, in practice we will only select a small number of members from this set, typically the first N solutions found, and so the weighting function controls which subset is calculated.

The solution found by the optimizer will only be optimal with respect to the set of candidate paths we generate. Indeed in some cases we may not be able to satisfy a demand at all because other demands have exhausted the capacity of the candidate paths generated for this demand. What should happen when we don't find a solution? There are a number of options open to us. We could increase N, the desired number of candidate paths, for all demands and rerun the optimizer. This would be time-consuming, although we could start the solver in a state representing the previous partial solution if the solver allowed this. It would require additional bookkeeping within the application to update the solver with this information. We could just search for additional paths for those demands that couldn't be totally satisfied. Assuming we could find such paths we would rerun the optimizer with this new set. Alternatively, we could just keep the solution for the demands that were satisfied and attempt to use CSPF on the residual network for the remaining demands. The current prototype uses none of these heuristics assuming, instead, that the network links are sufficiently lightly loaded that this scenario is unlikely to occur.

Even if all demands are satisfied, the choice of candidate paths may lead to a very non-optimal solution. Consider the network shown in Figure 2, taken from [Köhler and Binzenhöfer 2003]. With a single demand between every pair of nodes, there are 380 demands that need to be routed. We assume that all these demands have no QoS constraints, i.e. they represent best-effort traffic. This assumption, coupled with the high degree of connectivity between nodes in this example, makes the solver very sensitive to the choice of candidate paths. The following table illustrates the problem. The $N = 0$ row shows the average and maximum link utilization when the problem is optimized using an edge-based model, with "Kohler Average" [Köhler and Binzenhöfer 2003] as the objective function. The subsequent rows show the behavior of the optimizer when using a path-based approach for increasingly large sets of candidate paths.

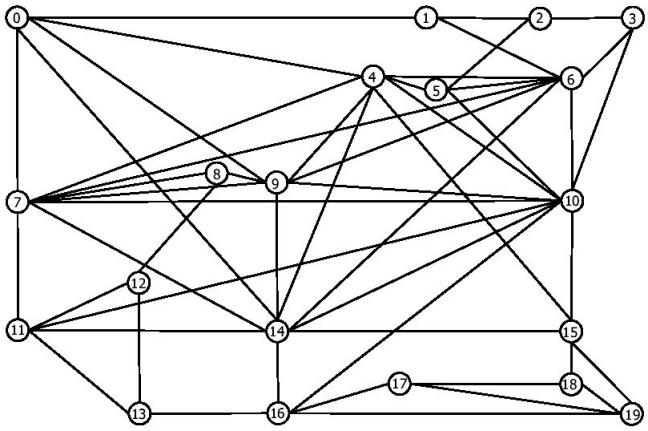

Fig. 2. A densely connected network topology

Maximum candidate paths, N	Average	Maximum	LSPs
Edge-based solution, 0	23.76	43.64	395
1	23.37	96.64	380
2	23.37	51.85	389
4	23.39	49.17	391
6	23.43	48.06	395
8	23.49	47.30	398
10	23.49	47.30	395
15	23.83	44.78	394
20	23.92	43.64	396
40	23.76	43.64	399
60	23.76	43.64	395

Consider the Maximum column first. We see that for very low values of N the path-based approach performs far worse than the edge-based one. Indeed, in some tests it was worse than an online CSPF approach. We have to generate 20 candidate paths for each demand before the maximum link utilization is comparable to the edge-based case. If we now look at the Average column we see that the value for $N = 20$ is still higher than the edge-based case. Choosing $N = 40$ fixes this, but the number of LSPs required to support these demands is still larger than the optimal solution. It is only when we reach $N = 60$ that the two approaches converge. As an aside, note the relatively small amount of demand splitting in the optimal solution, indicated by the number of LSPs, even though the particular optimization criteria being used does not penalize splitting.

Clearly in this example we should use an edge-based approach to solve the problem. The main disadvantage of such an approach, the difficulty in enforcing QoS constraints, does not apply here. In examples where the QoS constraints are a lot more restrictive the path-based approach works well. This is because the number of acceptable candidate paths will be quite small, and so beyond a relatively small value of N the candidate path set will not change.

For examples between these two extremes the best strategy is less clear, and we may need to resort to using heuristics. For example, we could choose a relatively small value for N, but then run an edge-based optimization to identify additional acceptable candidate paths to add to this set. If we find that all paths in the edge-based solution satisfy the QoS constraints then there is no need to run the path-based version at all.

In some cases the first N paths may be very similar to each other. Consider the example illustrated in

Figure 3. The first N paths we select might only differ by the first two hops, with the rest of the path

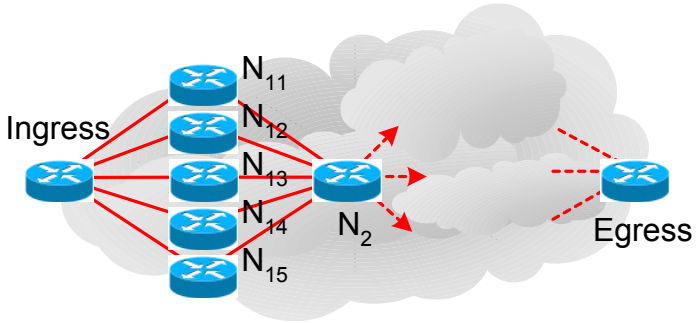

Fig. 3. Pathological candidate path example

being common to all the members of the candidate set. If the bottlenecks occur in the remaining segment of the path then we will not have any alternative routes, even though it looks like we have provided the optimizer with lots of alternatives. There are no really foolproof solutions here, other than a collection of heuristics. As mentioned earlier, we could use an arc-based solution to suggest paths, or generate more than N paths initially, followed by a filtering phase to promote "variety". But in pathological cases this may be counterproductive. Another approach might try to identify structural or hierarchical properties that might be exploited. But perhaps it is better to use such hints to decompose the whole optimization, rather than just for candidate path selection.

MPLS allows label-switched paths to be loose or strictly routed. In the loose case a partial route is specified, in the extreme case containing just the ingress and egress, and the system attempts to fill in the gaps. At present the TOAD prototype does not allow such routing constraints to be attached to demands. If they were added it would clearly complicate the candidate path selection process. For example, consider the case where a single intermediate point P is specified. We might Construct (up to) N candidate paths from I to P using the original QoS constraints, and then another N from P to E. We would then need to choose from the N^2 combinations the N paths with the smallest weight that satisfy the QoS constraints. This is clearly inefficient, and certainly doesn't scale when there are multiple intermediate node constraints. A better approach might be to develop a modified version of A*Prune that enforced these additional constraints as part of its operation. Of course we could take the view that loose specifications are only of use when an operator doesn't have access to an offline system, and so we can ignore these at the demand level.

A network topology, excluding intermittent faults, changes far less frequently than the demands over the topology. This suggests it may make sense to cache good routes in a database, giving us a head start when rerunning the optimizer. The candidate path selection process in the TOAD prototype is relatively unsophisticated at present, and therefore fast in comparison to the LP solver that has to process the resulting paths. The prototype therefore doesn't implement a path caching mechanism.

We now turn our attention to the case where there are multiple demands between two points, each with a different traffic class and priority. We could argue that we should route the strictest demands first, and then use the same paths for the more liberal demands to save time. Such a strategy might also lead to greater sharing of LSPs, using the EXP bits, in the provisioned solution. But we could also argue we need more LSPs for the more liberal demands with low priority, on the grounds that a lot of the bandwidth may have been consumed by other demands before we get a chance to route the low priority demands. It is hard to start the A*Prune algorithm in mid-flow, and so if we require additional paths we have to start from scratch again. This suggests a strategy where we generate more than N paths for the liberal demands and then pick the first N of these that satisfy the stricter constraints for the higher priority/stricter demand. At present the prototype doesn't attempt to optimize path selection for such cases, with each demand being treated independently.

This discussion also suggests a limitation of the current approach. Given traffic classes with different priorities, we optimize them in stages. But the candidate paths are computed for all demands in advance, and loaded into the optimizing component (see Section 6). With hindsight, it may make more sense to just compute candidate paths for those demands with the highest priority traffic classes. After these have been routed we then construct the candidate paths for the demands with the next highest priority traffic classes using the residual network bandwidth. This would have the advantage of not constructing paths for low priority traffic that are infeasible once the higher priority demands have been placed. Due to the low link utilizations typically encountered in real networks, this limitation may be more theoretical than real.

The TOAD prototype supports both edge and path-based demands, which raises the question of how the TOAD distinguishes between these cases. At present, any demand without an associated candidate path set is treated as an edge-based demand. Candidate path generation is triggered manually by the user, allowing the number of paths to be configured. Running the optimizer prior to this step results in all demands being treated as edge-based demands. A traffic class with a maximum hop count of zero will produce empty candidate path sets for all the demands associated with this class. We use this convention when we wish to use the edge-based strategy on a subset of the demands.

6. THE SOLVER

The primary role of the TOAD prototype is to solve multicommodity flow problems that arise as a result of optimizing demand placement. We could use heuristics to perform this task, as illustrated by [Lee et al. 2004]. However, our current approach is to use a linear program to solve the problem, relying on hierarchical decomposition to make the solution scaleable. Chapter 17 of [Ahuja et al. 1993] provides a good introduction to the general multicommodity flow problem. At this stage in the discussions the reader may find it helpful to refer to Figure 4, where an overview of the TOAD architecture is presented.


Those readers just wishing to get an overview of the TOAD can skip over the remainder of this section. It contains a lot of details that may be hard to follow without some prior knowledge of linear programming, modeling languages, and the multicommodity flow problem

6.1 MPL

The TOAD prototype currently uses the MPL modeling language from Maximal[Maximal 2004], with XA as the underpinning solver. Quoting from Maximal's web site, "The MPL Modeling Language offers a natural algebraic notation that enables the model developer to formulate complex optimization models in a concise, easy-to-read manner. Among modeling languages, MPL is unrivaled in its expressive power, readability, and user-friendliness. The MPL Modeling Language was designed to be very easy to use with a clear syntax making the process of formulating models in MPL efficient and productive. MPL is a very flexible language and can be used to formulate models in many different areas of optimization ranging from production planning, scheduling, finance, and distribution, to full-scale supply-chain optimization." Unfortunately, MPL doesn't always live up to these expectations ...

There are many different models we might wish to use in a TE optimizer, depending on the exact problem we are trying to solve, and the optimization criteria we wish to use. Rather than hard-wiring all these decisions into the application, the TOAD allows part of the model to be imported via a strategies file. This file, using XML format, contains one or more templates, and one or more strategies based on these templates. Many strategies are similar to each other, just differing in the objective function. The templates factor out those aspects of the model common to a set of strategies. So a typical strategies file would look like

```
<?xml version="1.0" encoding="utf-8"?>
<strategies xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
            xsi:noNamespaceSchemaLocation='Strategies.xsd'>
  <template name = "DemandPlacement" ... />

  <strategy
```

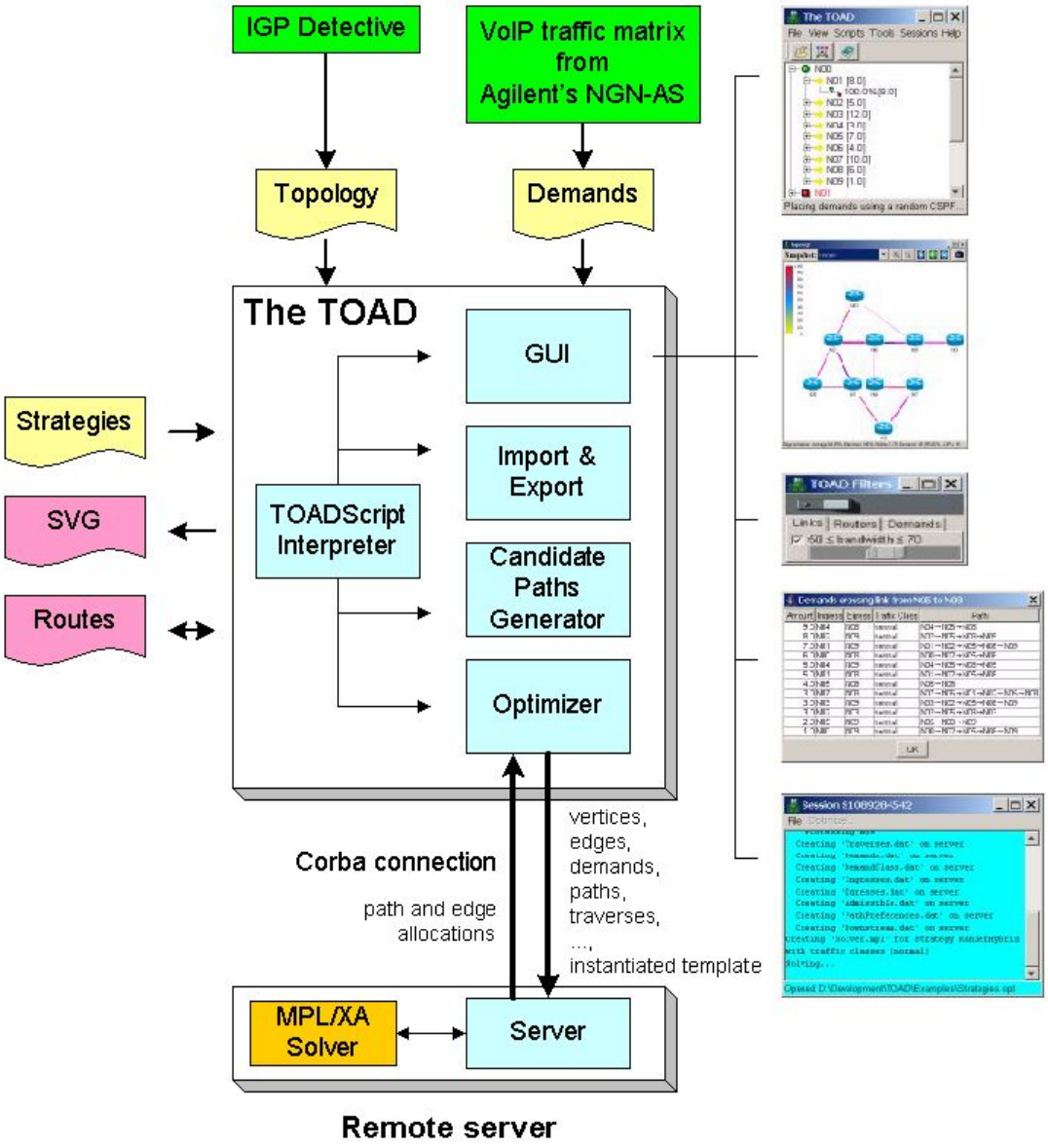

Fig. 4. An abstraction of the TOAD architecture

```
  name = "TotalDeficit"
  template = "DemandPlacement">
  ...
</strategy>

<template name = "AdditionalBandwidth" ... />

<strategy
  name = "AdditionalBandwidthLinks"
  template = "AdditionalBandwidth">
  ...
```

```
  </strategy>
</strategies>
```

We start by describing the structure of the templates, and then define some example strategies.

6.2 Templates

An MPL model for optimizing path-based demands minimising routing costs would look very similar to one that minimized the maximum link utilization. In both cases many of the constraints would be identical. We would like to factor out the common aspects of these problems, allowing each strategy to not be obscured by the common infrastructure. Most of the instance-specific features can be separated off into separate data files that can be loaded by the solver. However, a few aspects of the problem, such as the number of vertices for example, cannot be treated in this way. We therefore introduce templates, i.e. chunks of MPL containing metavariables that are instantiated each time we solve a problem instance.

The TOAD currently supports two kinds of templates. The first, a "demand placement" template, is used to compute a set of routes for the demands. The result of the optimization process is a bandwidth for each candidate path, in the case of a path-based demand, and a bandwidth for each (demand,edge) pair for edge-based demands. But there are other related problems we may be interested in. For example, suppose there is insufficient bandwidth in the network to satisfy all the demands. We may wish to know which links would need to be increased in capacity, and by how much, in order to satisfy these demands. Altering links is expensive, and so we might also wish to know what are the minimal set of links that would need to be altered to satisfy the demands. The results of optimizing such a problem are rather different to the demand placement case, and we use an "additional bandwidth" template for this situation. Additional template types may be added in future, as the need arises.

The model eventually passed to the solver is constructed from three sources; the strategy being used, the template on which this strategy is based, and details of the particular problem being optimized, such as the number of nodes and edges. The template body is an MPL program, extended with a set of metavariables that are expanded out prior to processing by the solver. We won't go through an example template in its entirety, although Appendix A provides sufficient details to give the reader a flavor of what they look like. To fully appreciate all the details of a strategies file would require some familiarity with MPL, and its syntax; providing this level of detail is outside the scope of this report. The appendix can therefore be safely skipped by most readers.

6.3 Strategies

We have talked about optimizing the placement of demands, but in reality there are a number of different criteria we might be interested in, each of which could be optimized. Examples include routing delay, routing cost, number of paths, maximum link utilization, and average link utilization. In many situations we will require some tradeoff between these potentially conflicting objectives, in other words a solution that balances two or more of these objectives in a weighted fashion.

In cases where all the demands cannot be satisfied simultaneously we may wish to know the amount of additional bandwidth required to satisfy these demands. Upgrading lots of links will be disruptive and expensive, and so we may wish to know the minimum number of links that would need to upgraded in order to satisfy a demand.

Some QoS classes, and hence demands, may have a higher priority than others. By this we mean that they should be more likely to be assigned an optimum route, and also be more likely to be satisfied, than those demands with lower priorities.

We could hardwire all these different cases into an application. However, the set of all possible strategies is hard to predict in advance. The best strategy to use will typically depend on the particular operating environment. A tool like the TOAD could be tasked with just generating a collection of data tables, with the model itself stored separately in a user-editable file. Whilst flexible, such an approach has a number of deficiencies. A user has to have detailed knowledge of a modeling language, and it would be easy to break the system. Furthermore, the application also has to be able to load back any solution and display it in a more user-friendly form, suggesting that at least part of the model has to remain fixed. Finally, some

strategies may appear complex when written in the modeling language, but be conceptually very simple. A strategy consisting of a hybrid of other strategies is a good example, or the encoding of a piece-wise linear cost function. We therefore adopt the approach of describing strategies within our XML format strategies file, allowing us to impose some structure on each strategy, with the TOAD performing some of the tedious work involved in translating the strategy into a valid MPL fragment.

We start with a simple example, the 'Kohler Average' optimization criteria taken from [Köhler and Binzenhöfer 2003].

```
<strategy
  name = "KohlerAvg"
  description = "Minimize average link utilization"
  prioritize = "true"
  template = "DemandPlacement">

  <objective>
    Value[KohlerAvg]
  </objective>

  <constraint><![CDATA[
    avg_utilization:
      Value[KohlerAvg] =
      SUM(edge: IIF(bandwidth=0, 0, Allocated / bandwidth));]]>
  </constraint>
</strategy>
```

Each strategy has a name, which is used for menus, and a description which is used in tool tips. The strategy specifies which template to use, and also whether the strategy should reflect traffic class priorities. A strategy must define an objective, which in the TOAD framework typically consists of the value of a decision variable, together with one or more constraints that bound this variable.

Here is a more complex example, illustrating the introduction of additional variables and constraints. This strategy corresponds to 'Kohler Hybrid', again taken from [Köhler and Binzenhöfer 2003].

```
<strategy
  name = "KohlerHybrid"
  description = "Kohler hybrid"
  prioritize = "true"
  template = "DemandPlacement">

  <variable> averageUtilization </variable>
  <variable> maximumUtilization </variable>

  <objective>
    Value[KohlerHybrid]
  </objective>

  <constraint><![CDATA[
    hybrid:
      Value[KohlerHybrid] =
      1000 * maximumUtilization + averageUtilization;]]>
  </constraint>

  <constraint><![CDATA[
    avg_utilization:
      averageUtilization =
      SUM(edge: IIF(bandwidth=0, 0, Allocated / bandwidth));]]>
```

```
  </constraint>

  <constraint><![CDATA[
    max_utilization[edge]:
      Allocated
      <= maximumUtilization * bandwidth / 100.0;]]>
  </constraint>
</strategy>
```

The KohlerHybrid strategy attempts to balance two different optimization criteria, the maximum and the average link utilization, in a single strategy. In some cases it may be preferable to define each strategy independently. We then combine the individual objective functions into a single function by using a weighted mean. However, due to possible differences in their scale, the objective functions need to renormalized by a factor, as discussed in [Erbas and Mathar 2002]. Hybrid strategies can be encoded in the strategies file using the following format.

```
<strategy
  name = "HybridExample"
  description = "Hybrid"
  prioritize = "true"
  template = "DemandPlacement">

  <strategy name = "RoutingCost" weight = "1"/>
  <strategy name = "LoadBalance" weight = "1"/>
  <strategy name = "Paths" weight = "1"/>
</strategy>
```

In this example the model would first be run three times. For each run only one of the objective functions is minimized, without taking the others into consideration. Each objective function is evaluated at each solution, and these are then used to calculate the appropriate normalization factor for each function. Finally, we solve the model using an objective function built from the sum of all three objectives, each one being multiplied by its normalization factor. This will clearly be more time consuming than combining optimization criteria in a more ad hoc fashion, as was done in the KohlerHybrid strategy. However, the structure of the strategy is clearer in the pure hybrid example, and the tradeoffs between each substrategy are easier to see.

We could handle traffic class priorities in a number of different ways. For example we could just use the priority as a weighting on the cost functions. If the routing cost of a path was weighted by the priority of the demand crossing the path, then the objective would tend to place higher priority demands over the shorter routes. However, this complicates the interactions between objective functions. It also gives us no guarantees that the system won't satisfy a lower-priority demand at the expense of a higher-priority one; it just makes it less likely. An alternative approach is to use stratification. We place all the demands with the highest priority first. We then solve those of next-highest priority using the residual network, and continue this process until all the demands have been placed. This approach is not necessarily optimum, as it ignores subsequent demands of lower priority during each optimization step. There may be two solutions to placing the highest priority demands with the same cost. But one solution may allow the remaining demands to be placed optimally, whereas the other one may only allow suboptimal solutions. However, given the objectives we are likely to use, this strategy should work acceptably, and is the approach adoped by the TOAD.

In many situations we may wish to use a non-linear cost function. For example, the cost of using a link should be more when the link utilization is reaching 100%. Such a cost function would tend to spread the load more evenly. But using non-linear functions moves us out of the realm of LP systems into more exotic, and potentially slower, solvers. A compromise is to describe such cost-functions by piece-wise convex linear functions. These are often quite adequate for our needs, and a typical example is described in [Erbas and Mathar 2002]. To code up such functions we just enumerate the points where the gradient changes. The TOAD ensures that these points define a valid function, and then uses this information to construct an appropriate set of constraints. Ideally, it would be useful if graphs such as these could be displayed and

altered within the TOAD. This would provide a convenient mechanism to allow the user to alter the exact form of a load balance cost function, and then rerun the optimizer, for example. At present, the user has to edit the functions manually, using the following syntax.

```
<piece-wise codomain="theta" domain="lambda" parameters="[edge]">
  (0, 0)
  (1/3, 1/3)
  (2/3, 4/3)
  (9/10, 11/3)
  (1, 32/3)
</piece-wise>
```

This example would be converted into the following MPL constraints, after checking that the points formed a convex surface.

```
theta_1[edge] : theta >= lambda;
theta_2[edge] : theta >= 3lambda - 2/3;
theta_3[edge] : theta >= 10lambda - 16/3;
theta_4[edge] : theta >= 70lambda - 178/3;
```

The C++ version of the TOAD prototype allowed the strategies file to be viewed and edited from within the application. The Java version updates the menus to reflect the optimization strategies available when a stragegies file is imported. However, these files have to be edited outside the TOAD environment.

6.4 CORBA server interface

Ideally you would like to run the solver directly from the TOAD application. However, this is not always feasible. Your desktop machine might not be powerful enough to run a large LP problem. You might also have more desktop machines than licenses. The TOAD prototype therefore adopts a client-server approach. The client communicates with the server via a CORBA interface. The interface is currently extremely simple, and is included in Figure 5. The TOAD client uses the interface to transfer model files, and the auxiliary

```
module Solver {
  interface OutputFile {
    void write(in string text);
    void close();
  };

  interface InputFile {
    string read();
    void close();
  };

  interface Session {
    string name();
    OutputFile openOut(in string fileName);
    InputFile openIn(in string fileName);
    void solve(in string modelFileName);
    void close();
  };

  interface Manager {
    Session createSession();
    void exit();
  };
};
```

Fig. 5. The IDL for communicating with the LP server

index and data files, to the server. It then requests the server to run the LP solver. After completion the client can request the contents of the various files produced as a byproduct of the solving process.

Ideally, the CORBA server would embed the solver directly inside it, using OptiMax in the case of MPL. However, at present we do not have a copy of this library, and so we call MPL as an external process. This difference is largely unnoticable to clients except that it doesn't fail as gracefully as you might like, and there is no mechanism for halting the process prematurely and retrieving a partial solution.

In our particular test environment the link between client and server is very slow. But even under more favourable environments the data files describing the demands, paths and so on may be tens of megabytes in size. We frequently wish to run the solver multiple times, with different optimization goals, in order to find a compromise between the various potentially conflicting goals. Transferring the data files to the server each time would be prohibitively expensive. The server therefore supports a session model. Clients create sessions, upload data files to the session, call the solver multiple times within the context of a session, and download results from the session. When the client terminates a session the files attached to the session are deleted.

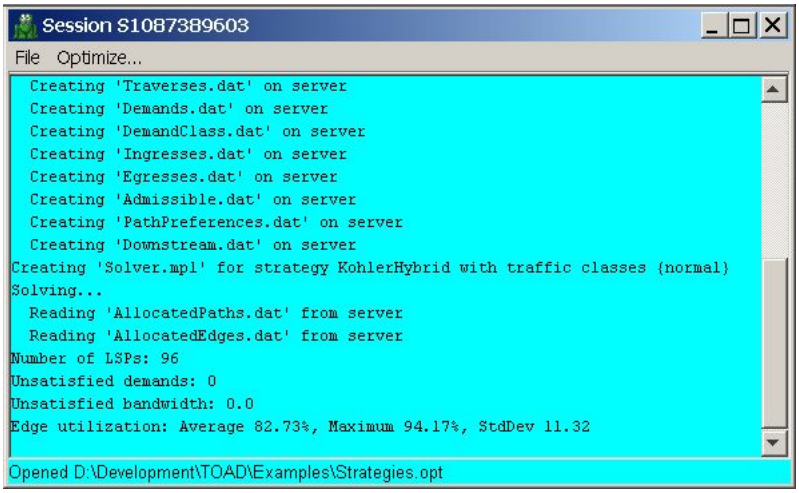

Fig. 6. The session window

7. CONVERTING ARC FLOWS TO PATH FLOWS

The final target of the optimization process is a collection of LSPs, together with the QoS constraints and bandwidth reservations for each one. In the case of a path-based demand the mapping to LSPs is straightforward. Each candidate path with a non-zero amount of bandwidth allocation gets mapped into an LSP. But what about arc-based demands? We know from the flow decomposition theorem[Ahuja et al. 1993] that any non-negative arc flow can be represented by path and cycle flows. The proof of this theorem gives us an algorithm for translating the flows across arcs into flows across a set of paths from ingress to egress. Whilst, in theory, the algorithm might also identify a set of cycle flows, we would not expect to encounter such a scenario in our situation. The objective functions used will penalize unnecessary link usage, and so the presence of a cycle in the arc flow would lead to a non-optimal solution. Alternatively we could use the two-stage process suggested by [Lee et al. 2004] to deal with these cycles.

The mapping from arcs to paths is not unique. Ideally you want to minimize the number of paths generated. In practice there tends to only be a single path that is generated in most cases, and so this simplistic approach may be sufficient.

Having converted the arc flows to path flows we check to ensure that each path satisfies the QoS constraints, and discard any that don't. This approach is satisfactory for demands with liberal traffic constraints, such

as best-effort traffic. If demands with strict QoS constraints were handled using the edge-based approach then we may find there are many demands that are only partially satisfied. We could attempt to enforce the QoS constraints within the model itself, as discussed in [Mitchell 2004a; Lee et al. 2004]. However, in these cases it may be preferable to use the path-based approach, perhaps using the successful paths identified by the edge-based solution to seed the candidate path sets. The current version of the TOAD does not implement this refinement.

8. SNAPSHOTS AND SVG

Having run an optimization the user wants to see some results. The TOAD displays both a tree view of the demands, and a graphical view of the network topology. Examples of these views are shown in Figures 7 and 8. Having completed a run of the optimizer the links in the topology view are color coded based on the current link utilization. A summary of the current solution is also shown at the bottom of this window. The tree view displays each router, and the demands originating at each router. In the case of a path-based demand there will be a set of paths associated with the demand, and the optimizer will indicate how much of the demand should be carried over each path. In the case of an edge-based demand the optimizer calculates a set of paths corresponding to the edge flows produced by the solution, and displays these paths as children of the demand.

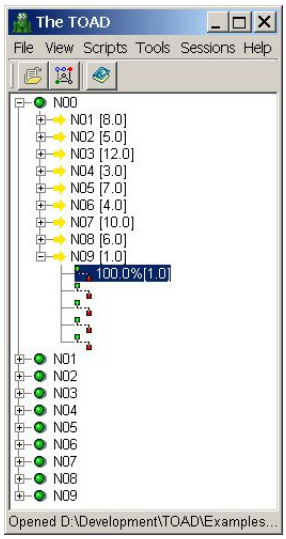

Fig. 7. The demand view

Whilst these approaches may be sufficient for visualizing a single solution, they are not adequate when you wish to see the changes that occur across multiple solutions. For example, you may wish to run multiple optimization strategies to see the different tradeoffs. Or gradually increase the demand bandwidth requirements to see the projected effect on the network links. Each time the optimizer is run the topology and tree views are updated to reflect the current solution, losing the previous state. However, the TOAD provides a simple snapshot facility, allowing the current state to be recorded, with an optional name. Furthermore, the prototype allows sequences of snapshots to be replayed like a simple movie, allowing trends to be rapidly identified.

Having run a sequence of experiments we would like to export the results in various forms. The TOAD can export the snapshot movies in SVG format, allowing them to be embedded and replayed from within a browser. Figure 9 illustrates this feature. Although not immediately obvious from a static screenshot, clicking the various snapshot names on the left hand column causes the topology to change to reflect the state corresponding to that snapshot. Furthermore, clicking on the camera icon displays a simple movie

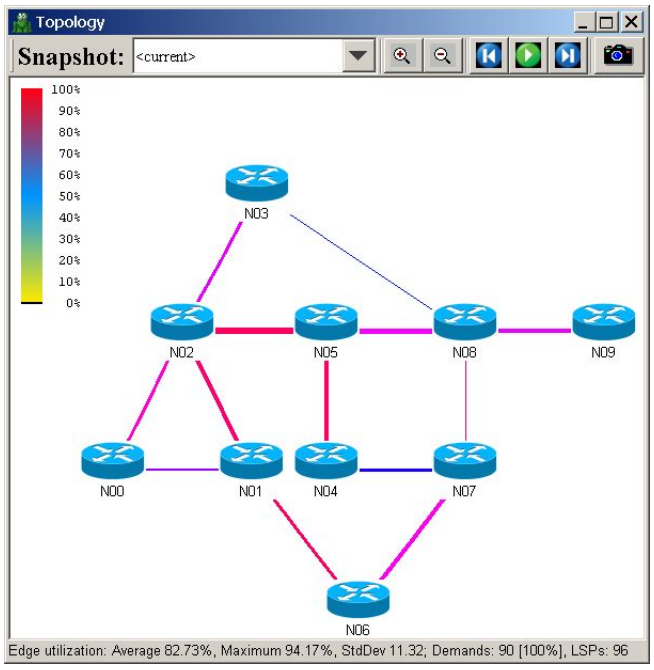

Fig. 8. The network topology view

that iterates through all these snapshots. If each snapshot was stored independently of all the others in the SVG file these files could get very large. However, the TOAD generates a combination of SVG elements, and JavaScript code to generate additional SVG elements from JavaScript data, resulting in comparatively small SVG files.

There are many other ways of visualizing and exporting the data produced by multiple optimization runs. The TOAD can also export charts summarizing link utilization statistics, again using SVG format. An example is presented in Figure 10. There is one column for each snapshot. The lower part of the column represents the average link utilization for the solution, whilst the upper part represents the maximum utilization. The black line at, or near, the top of the column records the percentage of demands that were satisfied in the solution. The particular example illustrated in the figure used a random constrained shortest path first approach to place the demands, and gradually increased the bandwidth of each demand. In the "+ 10%" snapshot we can see that not all the demands could be routed satisfactorily, presumably because of an unfortunate ordering of the demands.

The results of an optimization run can be exported in a routes file, an example of which is shown below.

```
<routes
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:noNamespaceSchemaLocation='Routes.xsd'>
  <ingress name='San Diego'>
    <egress name='Argonne'>
      <class name='Video'>
        <path bw='10'/>
        <path bw='20'>
          <hop name='Houston'/>
        </path>
        <path bw='15'>
          <hop name='Palo Alto'/>
        </path>
```

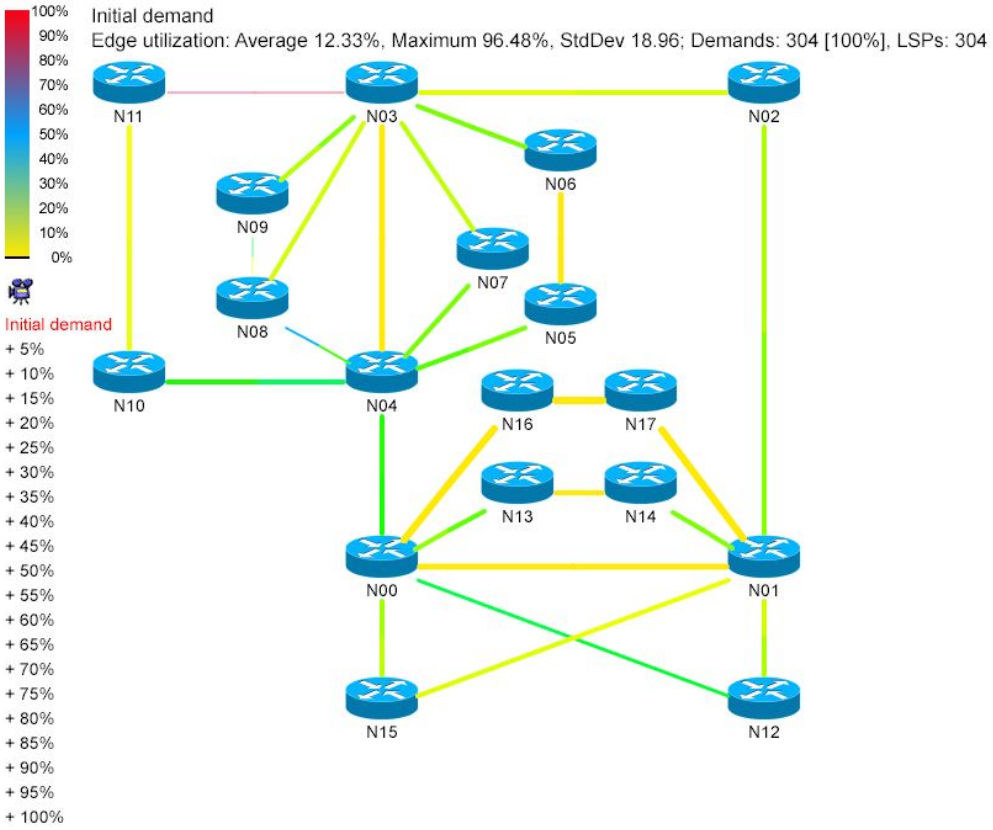

Fig. 9. An SVG file displaying multiple snapshots

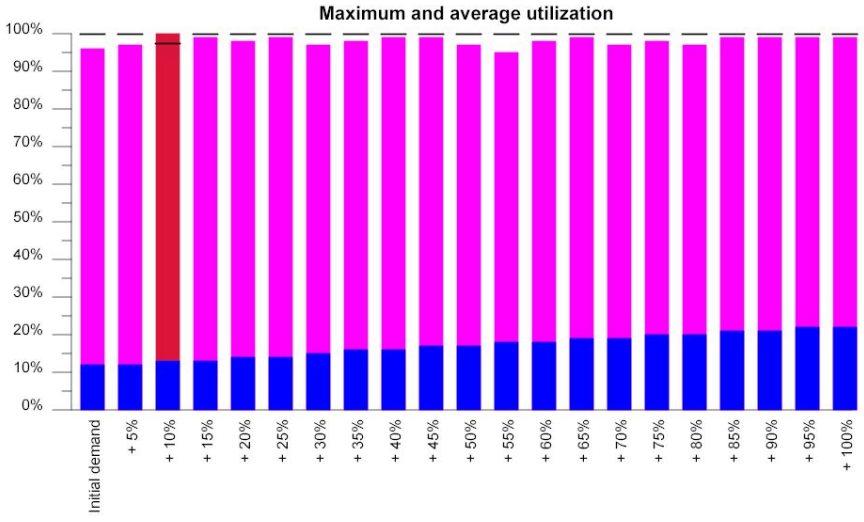

Fig. 10. An SVG chart for multiple snapshots

```
</class>
<class name='BestEffort'>
```

```
        <path bw='5'>
          <hop name='Houston'/>
          <hop name='Atlanta'/>
          <hop name='College Park'/>
          <hop name='Princeton'/>
        </path>
      </class>
    </egress>
    ...
  </ingress>
  ...
</routes>
```

The routes file could be processed by a provisioning system, to deploy this solution. The file can also be read back into the TOAD, allowing the set of candidate paths to be reused in subsequent runs of the optimizer.

9. TOADSCRIPT

Whilst the basic TOAD GUI may be sufficient for one-off experiments, there are also situations where a more automated approach is required. For example, suppose you are interested in a capacity planning scenario where you want to gradually increase the demands between a set of endpoints and see the effects on the network links. Clearly you could perform these steps manually, but it would be very tedious. A better solution might be to write a script to perform this task. The TOAD prototype has a JavaScript/EcmaScript interpreter embedded within it[Lugrin 2004], specialized for running TOAD-related tasks. Using this TOADScript interpreter it is possible to automate many repetitive tasks. Figure 11 gives an example script that gradually increases the bandwidth requirements of every demand, routing them using CSPF routing at each step. The resulting animated topology view, and link utilization chart, are saved in SVG files for later viewing. It was this script that was used to construct the SVG files shown in Figures 9 and 10. The same approach can be used to explore the effects of link and node failures, different optimization strategies, and a whole host of related problems. It is possible to construct simple dialogs using TOADScript, and scripts can be added to the menu system. These features allow the TOAD application to be customized for individual users.

10. FILTERS

Although the snapshot mechanism allows you to visualize course trends, this, by itself, is not sufficient. For example, consider optimizing a problem using two different optimization strategies. We may find that almost all the links and demands are largely unaffected by the choice of strategy. However, there may be a handful of links, or demands, that are treated very differently by the two strategies. In these cases we might like to suppress the display of links whose utilization differed only slightly between snapshots. In other cases we might only wish to display links or routers of a particular type, or links with a utilization or bandwidth within a specified range. The TOAD implements a powerful filtering mechanism, illustrated in Figure 12, to support these visualization goals. The topology and demand file formats are extensible, in that arbitrary attributes can be attached to nodes, edges and demands. The filtering mechanism is adaptive, allowing the user to filter on the particular attributes in use for the particular topology and demand files currently loaded.

11. PERFORMANCE

In this section we explore the performance of the TOAD and its optimization strategies. The current implementation of the TOAD uses the MPL modeling language. This language has no published grammar, and an annoying habit of looping, hanging, crashing, or reporting content-free error messages whenever it encounters models it doesn't like. When coupled with the lack of a support contract, this makes it difficult to experiment with different formulations of the problem efficiently. The results presented in this section reflect the performance of the current models, but it is unclear to what extent the performance could be improved

```
base = "...";
TOAD.load(base + "Kohler18.net");
TOAD.load(base + "Kohler18.dem");

function scaleDemands(percent) {
  for (r in TOAD.network.routers) {
    for (d in r.demands) {
      d.bandwidth += (d.bandwidth * percent / 100);
    }
  }
}

function unscaleDemands(percent) {
  for (r in TOAD.network.routers) {
    for (d in r.demands) {
      d.bandwidth = (d.bandwidth * 100) / (100 + percent);
    }
  }
}

TOAD.placeDemandsUsingCSPF();
TOAD.snapshot("Initial demand");

for (i = 5; i <= 100; i += 5) {
  scaleDemands(i);
  TOAD.placeDemandsUsingCSPF();
  TOAD.snapshot("+ " + i + "%");
  unscaleDemands(i); // Reset to base level
}

TOAD.saveSVG(base + "Results/Test.svg");
TOAD.saveSVGChart(base + "Results/TestChart.svg");
```

Fig. 11. An example TOADScript

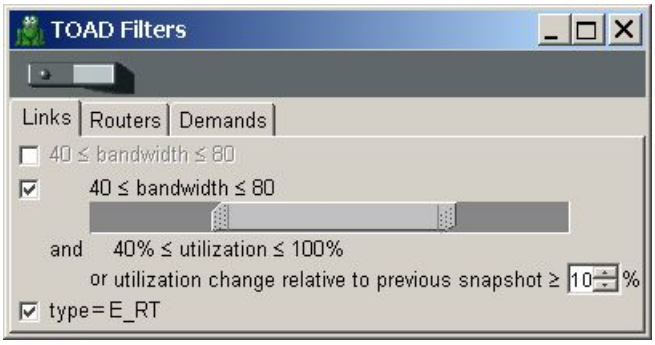

Fig. 12. An example filter panel

by reformulating the problem, or using a more robust modeling language. The TOAD is also currently hindered by the need to ship the data to a remote machine, over a slow connection. The timings presented reflect the total elapsed time, including these connection overheads. For small examples the networking time will tend to dominate. In a real implementation we would embed the solver within the application, avoiding these overheads.

With these caveats out the way, we start by considering the tradeoff beween using a path- and edge-based approach. Figures 13, 14, 15, 16 and 17 show the performance of the TOAD using three different strategies on a variety of topologies. The Kohler Average and Hybrid strategies are taken from [Köhler and

Binzenhöfer 2003]. As the names suggest, Kohler Average attempts to minimize the average link utilization, whilst Kohler Hybrid balances the desire for a small link utilization with the requirement to minimize the maximum link utilization. The Simple Hybrid strategy is defined by

```
<strategy
  name = "SimpleHybrid"
  description = "Simple routing cost/load balance hybrid"
  prioritize = "true"
  template = "DemandPlacement">

  <strategy name = "RoutingCost" weight = "1"/>
  <strategy name = "LoadBalance" weight = "1"/>
</strategy>
```

where RoutingCost attempts to minimize the overall routing cost by weighting the amount flowing over each edge with the cost of using the edge, and LoadBalance implements the load balancing metric defined in [Erbas and Mathar 2002]. The example topologies and demands are taken from [Köhler and Binzenhöfer 2003]. The dotted lines in each graph show the results obtained by using an edge-based strategy, for comparison with the path-based results. The numbers above each point in the graph record the number of LSPs that were required for that solution. In a few cases the solutions were only partial, with some of the demands being only partially satisfied. These cases are indicated by a red LSP count. The demand set used in these experiments has a single traffic class, representing best-effort traffic. We might expect that this would favour an edge-based strategy. However, the graphs illustrate that for many of these examples a relatively small candidate path set is sufficient to yield a solution close to the edge-based version. Furthermore, as shown in Figure 18, the path-based approach can produce these results quickly, at least for small examples. It is only when we reach the Kohler20 example, where the topology is very stongly connected, that we start to see the deficiencies of a path-based approach. In this example there are a large number of distinct paths between each pair of nodes, and so we may need to generate a large number of candidate paths to ensure our solution can use the optimum ones found by an edge-based approach. This example illustrates why it may sometimes be useful to use the edge-based approach to seed the candidate path set, although this is also time-consuming.

The graphs show that the Kohler Hybrid strategy does a better job of balancing the maximum and average link utilization than the Simple Hybrid strategy. However, although not shown on these graphs, the Simple Hybrid strategy produces solutions with a smaller standard deviation. It is also easier to adjust the weighting between the average and the maximum components, and the cost function for link utilization, making the Simple Hybrid approach rather more flexible than the Kohler Hybrid. For comparison, the graphs also record the solution obtained by performing a simple "online" CSPF placement. These figures are slighly pessimistic, as an online algorithm could be smarter than our current implementation, taking into account the residual capacity of the links when placing demands. Nevertheless, even using these pessimistic figures, we see that the difference between an offline globally-optimized solution and an online incremental approach, is comparatively small when the average link utilization is considered. The difference is more marked when we example the maximum link utilization.

We now consider an example involving multiple traffic classes. We use the example from [Mitra and Ramakrishnan 2001]. There are four traffic classes:

```
<class name='Voice' id='0' priority='0' hops='3' delay='1000' cost='15'/>
<class name='Video' id='1' priority='0' hops='3' delay='1000' cost='20'/>
<class name='Premium' id='2' priority='0' hops='4' delay='3000' cost='10'/>
<class name='BestEffort' id='3' priority='1' hops='0' delay='3000' cost='5'/>
```

This example illustrates a deficiency in the current traffic class specification model. The Mitra paper defines the Voice and Video classes as having the "minimum hop length", whereas the TOAD only allows us to specify absolute QoS constraints. Introducing a relative hop limit would be a simple extension. From this traffic class specification we can see that the best effort traffic has a worse priority than the rest of the

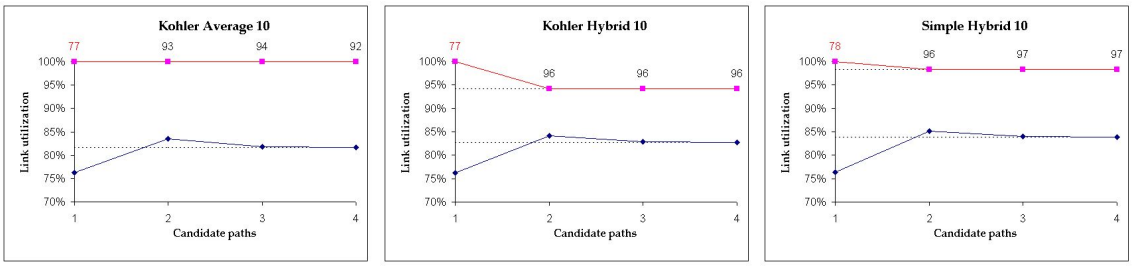

Fig. 13. Kohler10 example (CSPF LSPs: 86, average 83.07%, maximum 100%)

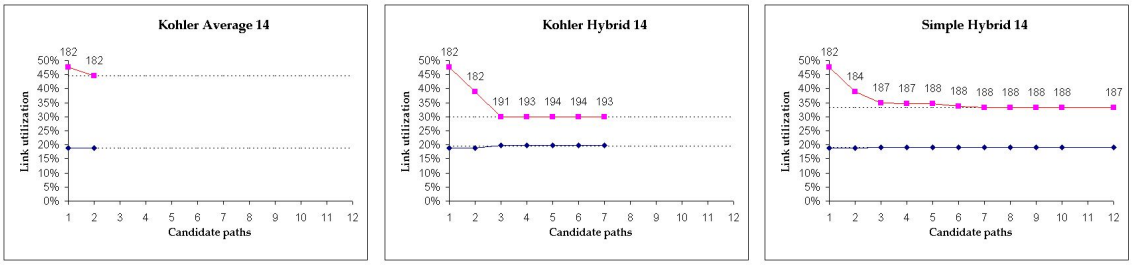

Fig. 14. Kohler14 example (CSPF LSPs: 182, average 18.94%, maximum 47.57%)

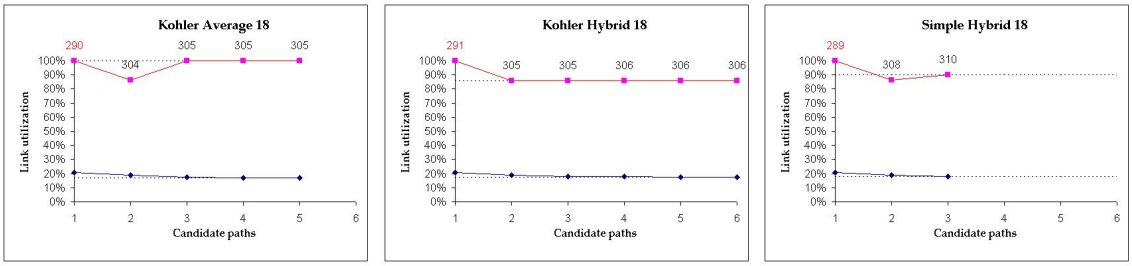

Fig. 15. Kohler14 example (CSPF LSPs: 304, average 22.66%, maximum 98.83%)

traffic. Furthermore, the hop limit of 0 forces the TOAD to use edge-based routing for this traffic class. The network in the paper is underprovisioned.

N	LSPs	Video	Voice	Premium	BE	Unsatisfied	Avg	Max	S.D.	Time
0	226	2	1	0	4	78350.0	73.42%	100%	19.07	7992
1	222	1	2	1	1	72735.0	76.72%	100%	16.13	6179
2	228	0	0	0	4	54800.0	76.01%	100%	19.17	6230
3	228	0	0	0	4	54800.0	75.79%	100%	19.75	6180
4	229	0	0	0	4	54800.0	76.08%	100%	19.24	6209

The table illustates the main deficiency of the edge-based encoding for such examples. Consider the first row in this table, representing the case where all demands are placed using the edge-based approach. The current model makes no attempt to enforce the QoS constraints for an edge-based demand. The system therefore routes the video and voice calls over edges that, when converted to a set of paths, lead to invalid solutions. In contrast, the path-based approach manages to route all the high-priority traffic, and nearly all the best-effort traffic. In the next table we double the link capacities to make the network loads more manageable.

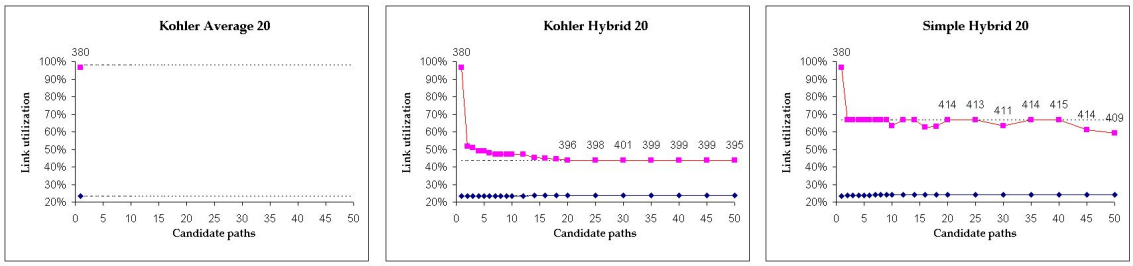

Fig. 16. Kohler20 example (CSPF LSPs: 380, average 23.37%, maximum 96.64%)

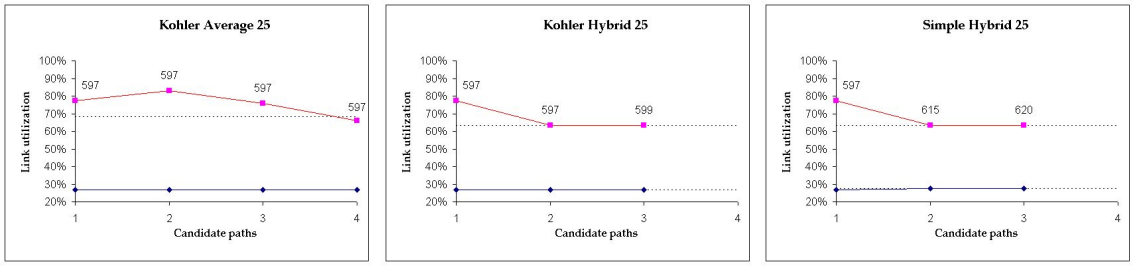

Fig. 17. Kohler25 example (CSPF LSPs: 597, average 26.88%, maximum 77.38%)

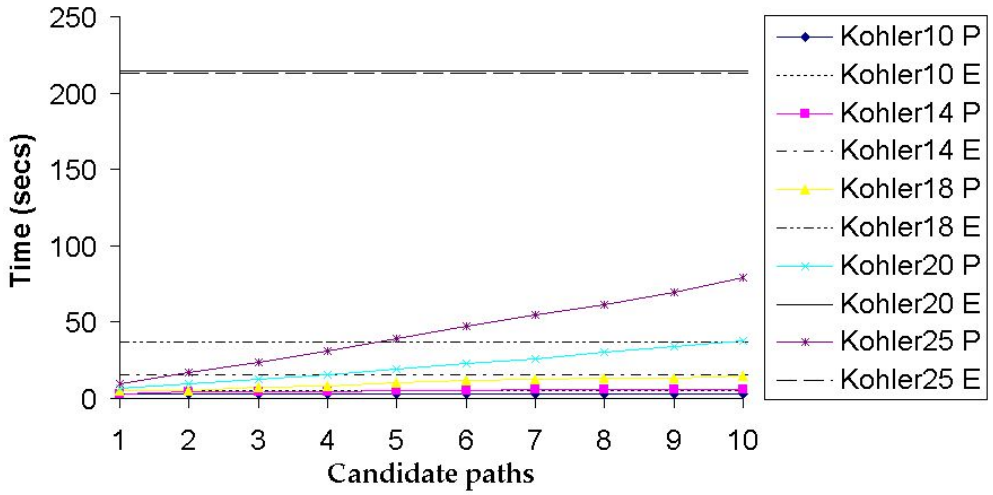

Fig. 18. Kohler timings

N	LSPs	Video	Voice	Premium	BE	Unsatisfied	Avg	Max	S.D.	Time
0	230	2	0	1	0	23550.0	36.99%	58.84%	14.19	7932
1	225	1	0	0	0	0	38.78%	75.11%	16.00	6169
2	231	0	0	0	0	0	38.32%	58.84%	14.22	6229

The edge-based approach again results in some invalid routes. The path-based approach quickly converges to an optimum solution as there are relatively few paths that can satisfy the QoS constraints of the demanding traffic classes.

There is no "best" strategy for solving a traffic engineering optimization problem. There are multiple conflicting goals, and the best trade-off will inevitably depend on the particular scenario and user preferences. The TOAD allows the user to add additional optimization strategies via the strategies input file. A number of different strategies have already been implemented, including

Routing Cost. Minimize the overall routing cost, by penalising links with long delays. Sum, over all edges, the amount allocated to the edge multiplied by the delay for the edge.

Kohler Average. Minimize average link utilization, [Köhler and Binzenhöfer 2003].

Kohler Max. Minimize maximum link utilization, [Köhler and Binzenhöfer 2003].

Kohler Hybrid. Balance between Kohler Average and Max, [Köhler and Binzenhöfer 2003].

Load Balance. Make cost of using a link get higher as link utilization increases, using a piecewise-linear cost function, [Erbas and Mathar 2002].

Paths. Minimize the number of paths used; only considers path-based demands.

Hybrid. An equal weight hybrid of Routing Cost, Kohler Average and Load Balance.

Figure 19 shows the different results produced by these stratregies, illustrating some of the tradeoffs that can be made.

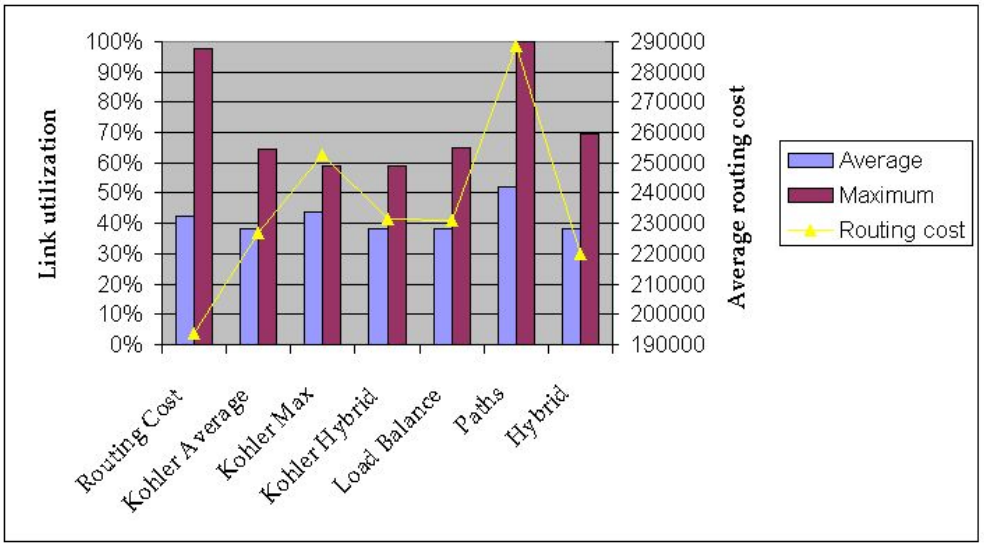

Fig. 19. The performance of different strategies on Mitra topology

12. FUTURE WORK

12.1 Tunnels and protection

One of the advantages of MPLS over a technology such as ATM is that it supports tunneling to an arbitrary depth through the use of the label stack. Tunneling can reduce signaling overhead, and reduce the state that needs to be held in the routers. In some cases it may also make it easier to provide backup routes as well, for example by just protecting a single tunnel rather than a set of paths that cross the tunnel. However, if the tunnel gets too big then it may be difficult to identify suitable backup paths. This raises the question of where should we place tunnels. There are two obvious strategies. We could either use the demands and topology to suggest good places for tunnels, and then perform a demand optimization as before. The tunnels will just appear as virtual links, and we can direct the optimizer to use these, in preference to the raw links,

by using appropriate cost functions. But such an approach has a number of obvious disadvantages. How do we determine the best position for the tunnels, and what bandwidth should we give them?

A more plausible approach involves first placing the individual paths, and then determining a collection of tunnels that can convey this traffic. There may be multiple ways of breaking up paths into tunnels, leading to another optimization problem[Menth and Hauck 2001]. Provisioning backup paths can also be problematic, as there is a trade-off. If we have lots of small tunnels then there will be a lot of signaling overhead, with state being maintained in each of the routers. We also have to determine backups for each of these tunnels separately. At the other extreme, we may be able to tunnel all of the original LSPs through a single LSP, at least for part of their route. But the bandwidth for this LSP will be large, and so if a link fails then we may have difficulties provisioning a backup path of the required size. This argues for the use of multiple smaller-dimensioned tunnels to simplify the path protection problem. The current TOAD prototype provides no explicit support for tunneling, although the methodology described in [Mitchell 2004b] might be used to suggest potential tunnels.

Network reliability is essential in core networks. Disruption can occur for several reasons, such as congestion, and link and node failures. MPLS LSPs can be protected at the path, node and link levels. The essence of path protection is in establishing an end-to-end backup tunnel for each of the primary LSPs. A backup LSP must typically be link and node-disjoint with its corresponding primary LSP. When a failure occurs the ingress router is notified of the failure, for example by RSVP-TE, and the traffic for the LSP is then rerouted to the backup LSP. The backup paths are typically computed at the time the LSP is initially provisioned. Numerous approaches to this problem can be found in the literature, e.g. [Sidhu et al. 1991; Bejerano et al. 2003; Grover and Doucette 2001]. Some approaches attempt to find primary and backup paths simultaneously, whilst others view the problem as a two-stage process.

The signaling delay from when a node or link fails, to the ingress being aware of this failure, will often be in the order of seconds. This is too long to satisfy many QoS constraints. Furthermore, the backup LSPs, if preprovisioned, will double the LSP state that needs to be maintained by each router. The essence of node and link protection is the establishment of local backup LSPs, or tunnels, that reroute all traffic around a failure. So, for example, to protect a link from n_1 to n_2 the system would establish an LSP from n_1 to n_2 that avoids this link. Since only a single, typically short, backup LSP is used for protecting a potentially large number of primary LSPs that use the link, there may be substantial savings in the amount of LSP state required. Furthermore, node n_1 can start using this detour as soon as it recognises the link has failed, causing a delay in the order of milliseconds rather than seconds. Of course the detour may cause the new path to violate the QoS constraints for the demand for a brief period, but this is typically preferable to a total failure for a few seconds. Once the ingress router becomes aware of the failure it can signal a new LSP that meets the tunnel constraints, and switch over to the new LSP if successful. The paths used by the link and node protection LSPs can be computed offline. Alternatively, some routers can be configured to find and provision acceptable backup paths automatically.

The TOAD does not currently address the backup path issue. The solutions computed by the TOAD can be protected using the automatic node and link protection facilities of the routers themselves, relying on CSPF online routing for any secondary backup paths that may be required.

12.2 Transition policies

The traffic demands that are fed to the TOAD represent projected demands for some future time period. Whilst they have their basis in reality, in that the predictions use previously obtained traffic matrices obtained from the network, they should not be confused with these traffic matrices. For example, as part of the prediction process we may smooth the raw data by pruning outliers, remove some demands entirely as they are too variable, and inflate the observed demands using trend analysis. So the demands fed to the TOAD for a particular optimization time period may be very different to the demands observed at any time in the past. The attributes attached to these demands should record the evidence used to produce these predictions, so that if we find a prediction is very different from what we observe during the period a solution is deployed then we can trace back to see how we arrived at these predictions. Information that is not relevant to the prediction, such as historical performance data, is therefore best left out of the demand

files, although it may still be worth collecting for other reasons.

Traffic engineering is an iterative process. Predicted demands are constructed from traffic matrices, and other trend data. We then determine a set of LSPs, and routes, to support these demands, and these LSPs are then deployed on the network. We then repeat the whole process again for the next time period. Of course this description is too simplistic in reality. In an extreme case we may end up completely changing the deployed LSPs every few hours, and no operator is going to be comfortable with such an upheaval to their network. This suggests the need for *transition policies* to control the acceptable degree of change between each optimization period.

Given a description of the currently provisioned set of LSPs, and a new set of demands, there are many different ways of satisfying these demands, each with a different impact on the existing network. Until now our description of the optimization process has completely ignored the current state of the network. This approach may lead to unnessary changes to the network, where an LSP supporting a demand is replaced by another LSP, with a different route, to support the demand at a later point in time, even though the original LSP would have been equally acceptable. At the other extreme, we could imagine trying to support the demands by making only the minimal changes to the existing solution. Here we trade off optimality for stability. For example, adjusting the bandwidth reservation of an LSP can be viewed as a fairly benign operation, and in some cases we may be able to adapt the existing LSPs to the new set of demands by just making such changes. Of course as time goes by the solution that results from making these minor updates may start to depart substantially from the solution that would result from an unconstrained optimization. This suggests that we should periodically relax our constraints, either after a fixed time period, or where the difference in optimimality between the solutions crosses a threshold.

We use the term 'transition policy' to refer to the rules controlling the freedom the optimizer has in making optimization decisions. A transition policy might

- give the optimizer complete freedom to choose the LSP routes,
- force the optimizer to keep the current LSPs, but to minimize the bandwidth allocation changes,
- identify important demands, and their LSPs, that shouldn't be altered if possible; where possible all changes should be confined to the remaining demands.

Clearly there could be many other examples of transition policy. Ideally the TOAD should be extended to allow such policies to be specified and enforced. Furthermore, there should be a mechanism for specifying what policy should apply at each transition point. As we have observed earlier, we may wish to use a very strict policy for most of the time, with a more liberal policy being used once a week, for example. This suggests we also need an administrative policy to control our use of transition policies, for example "only perform major reoptimizations once a week". This situation is illustated in Figure 20.

Given a sequence of demand sets for consecutive periods of time, the transition policy generator determines the transition policy that should be used for each run of the optimizer. This diagram also shows how the result of this process should be a collection of provisioning scripts. Each script updates the current configuration of LSPs, by changing bandwidths, adding and removing LSPs, to reach the next configuration. The traffic profile itself is produced from the traffic matrices, plus additional trending information. This process is illustrated in Figure 21. It is worth noting that the period over which the traffic matrices were generated, i.e. the duration spanned by Traffic Matrix 1..N, is totally independent of the prediction times ΔT1, ..., ΔTN. For example, we might use a large history to make predictions for a small amount of time ΔT, or use a small history to extrapolate a long way into the future.

Whilst the approach illustrated in these figures is conceptually simple, it also has some limitations. It seems reasonable to assume that management stations will be collecting performance data for the network, and that this data should be able to influence the optimization/provisioning process. There are two obvious things that could be influenced, these being the transition policies, controlling which demands should be reoptimized at each step, and the duration of each optimization period. It is difficult to support such an approach using the architecture from Figure 20. Once the transition policy generator has annotated the profile file then there is no opportunity to alter the transition policy between `Demands` Δ`T2` and `Demands` Δ`T3` in response to monitoring the performance during Δ`T1`, or to shorten the period Δ`T3`. To allow such changes we

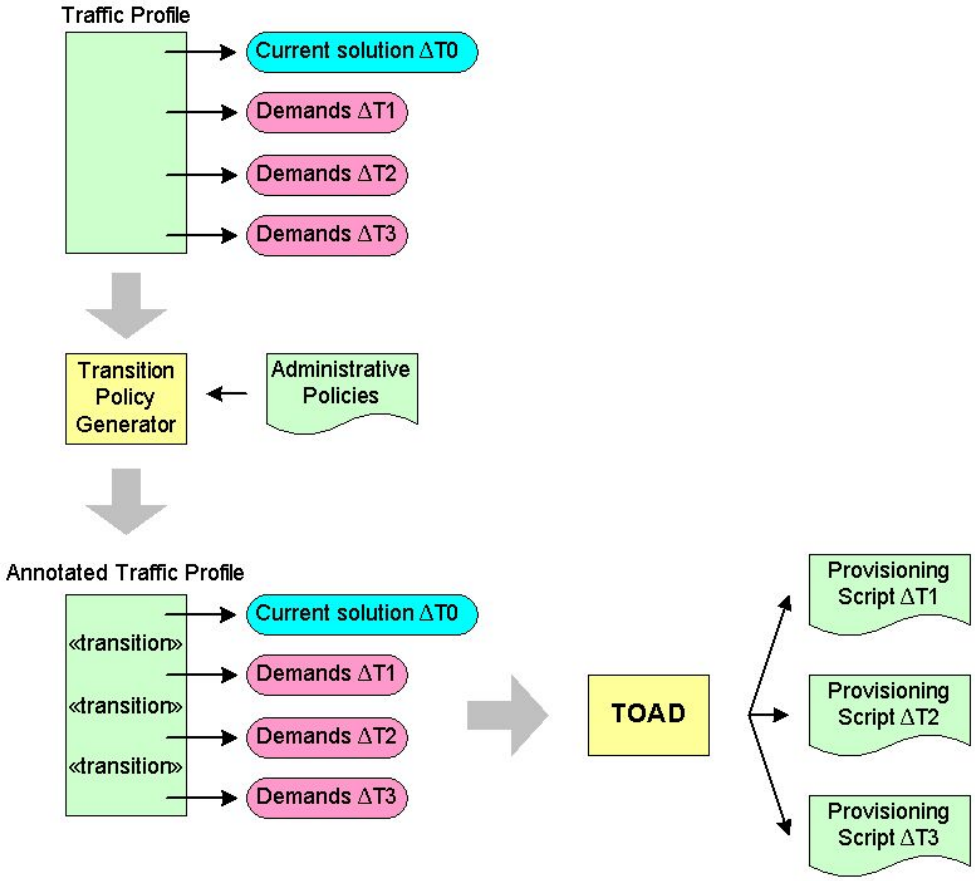

Fig. 20. An batch approach to transition policies

need to move to a model where the optimization for period $\Delta T3$ was being performed during period $\Delta T2$, where the transition policy has been chosen based on the administrative policy, perhaps in conjunction with performance data collected during period $\Delta T1$. This suggests we might eventually want to move towards the architecture shown in Figure 22.

13. CONCLUSIONS

In a pure IP network routers react to changes in the network topology by computing new paths. This has made the Internet an extremely robust communication network, even in the face of rapid growth and occasional failures. However, such adaptive behaviour does not ensure that the network runs efficiently. IGP metrics can be adjusted to migrate some traffic away from hotspots, but the ramifications of such changes are often difficult to predict, and frequently create as many problems as they solve. It can also be challenging to provide QoS guarantees without resource reservations.

MPLS gives the operator control of the routes used for the traffic, and can reserve resources along these routes. Of course this power comes at a price, in particular the need for a signalling protocol to establish the routes and reservations. Signalling protocols take time to establish an end-to-end connection, generate additional signalling traffic, and require state in each router. MPLS is therefore best suited to supporting long-lived aggregated traffic, rather than individual microflows. MPLS should not be viewed as a replacement for IP, but rather as a complement to it. The route chosen for an LSP can be left entirely to the routers, for example when provisioning an aggregated best-effort tunnel. In other cases the operator may attach

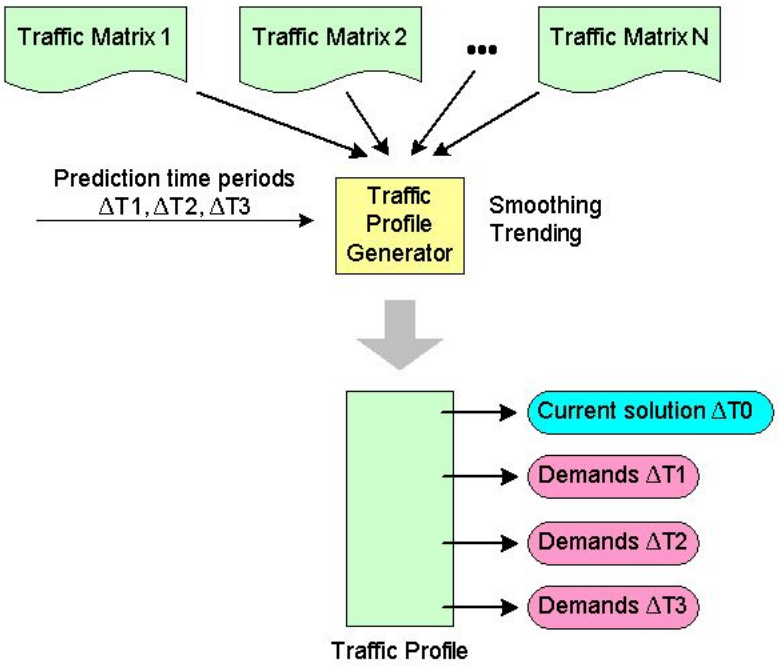

Fig. 21. Constructing traffic profiles

QoS constraints on acceptable routes, such as the total delay, or require the LSP to pass through one or more specified routers. Taken to extremes, the operator can even fix the exact route to be followed by the LSP. However, there is a penalty to be paid for such precision. If any link fails along this path then the MPLS tunnel itself will also fail, as no other paths will be able to match the specification. To address this issue, LSPs are typically specified with loose constraints, allowing some flexibility in the route, or with a secondary route specification that can be used as a backup, or a combination of these approaches.

Online CSPF routing can produce resilient solutions, together with acceptable QoS guarantees in many cases. It is therefore natural to ask whether it is worth performing a costly offline optimization to route these demands. Performing a global optimization, of the form described in this document, requires demands that are both stable and predictable. Of course even in the case of online LSP provisioning we require demands to have these attributes, but not to the same extent. In particular, we do not need to predict demand bandwidth requirements hours or even days in advance. Furthermore, in an online CSPF-based solution we can dynamically adjust the demand bandwidth requirements as the need arises, computing a new route if necessary. Whilst we can also use such techniques for LSPs that have been routed offline, we can no longer claim optimality after making such changes, undermining the justification for the offline approach.

It seems clear that an offline approach has stricter preconditions on the demands, and will often be less resilient than an online solution. In some cases these disadvantages may be offset by improved network utilization, making it important to quantify the potential extent of this improvement. However, it is difficult to make a fair comparison between the offline and online approaches. After all, even in the online case the operator can incorporate global knowledge, for example by partially specifying routes, or using affinity groups. Clearly we have a spectrum of possibilities, with online CSPF-based completely loose routing at one extreme, and a global optimization using totally pinned routes at the other end.

The disadvantages of the offline approach are fairly clear, namely the need to predict demands in advance, the potential fragility of the solutions, and the time required to run the optimizations. What are the advantages that can offset these problems? In a heavily loaded network an offline optimizer may be able to route more LSPs than an online one. This is because a CSPF solution, at least in its simplest form, takes

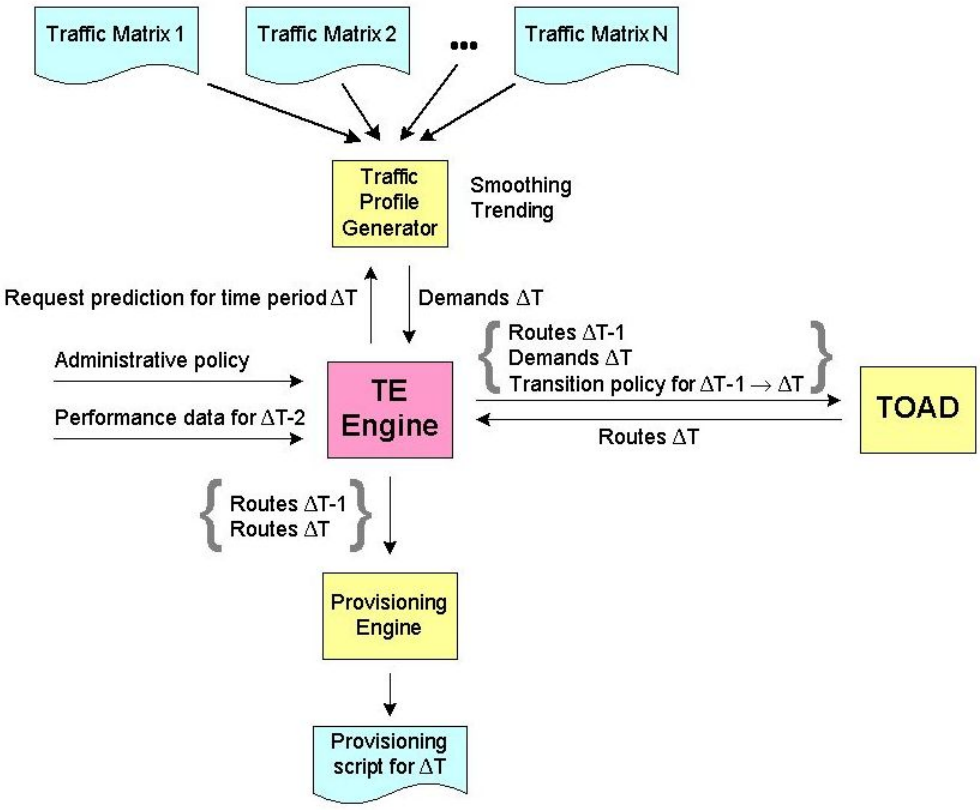

Fig. 22. An incremental approach to transition policies

no account of future requests, either real or predicted. Many MPLS networks will typically allocate some proportion of their capacity to LSPs, with best-effort traffic soaking up the spare capacity. In these scenarios the link utilization required to support the LSPs may be relatively small ($< 50\%$). Although CSPF routing, in extreme cases, may not be able to route as many LSPs as a globally optimised solution, this situation is unlikely to occur with such low average link utilizations. Of course a CSPF solution may still drive some links close to their maximum capacity, as illustrated in Section 11. Furthermore, these links, and their adjacent routers, may become sufficiently overloaded that the traffic flowing across these links is adversely affected. In extreme cases this may result in SLA violations, with their associated costs. However, without access to realistic cost models for networks, and SLAs, it is obviously difficult to quantify the extent of this advantage.

In addition to reducing the maximum link utilization, an offline approach will also reduce the average utilization in many cases. The results shown in Section 11 suggest that this improvement is comparatively small in many cases. However, small differences in average link utilization do not, of course, mean that the cost savings will only be negligible. In some environments the operator may be using expensive leased lines, and charged depending on the amount of bandwidth used. In such a setting a small change in utilization may still translate into a substantial saving. However, without access to realistic costing models it is difficult to validate such assertions.

The TOAD allows a demand to be split across multiple paths. Clearly in some cases this flexibility can be used to reduce link utilization by load balancing. Unfortunately, it is not always possible to spread the traffic across the LSPs without introducing additional packet reordering within the constituent microflows. We have argued that there are some contexts, for example a voice gateway collocated with an MPLS ingress router, where it may be possible to split a demand without splitting individual flows. Nevertheless, even in

this case demand splitting requires additional mechanisms to enforce the desired bandwidth split. We must therefore ask whether demand splitting is really worth the effort required to support it. One advantage of demand splitting is that it simplifies the LP model. Requiring our solutions to assign at most one path to each demand moves us into the realm of Mixed Integer Linear Programming (MILP), which can often be computationally more expensive to optimize. You might expect that the solutions computed by the TOAD would make extensive use of demand splitting. However, our test results show that very few demands are actually split in practice. The difference between the number of LSPs used in the best solution, and the number of demands, is typically quite small. Of course this does not mean that these additional LSPs are not beneficial; in some cases there could be a substantial difference in link utilization for one or more critical links. Nevertheless, when there are many demands, each with a relatively small bandwidth requirement, then splitting is rare, and has little impact on the optimality of the solution. Without access to representative demand and cost data, it is hard to draw any firm conclusions about the benefit of such a facility in the real world.

The TOAD prototype supports multiple egresses via virtual links and nodes. The initial motivation was to support multiple voice gateways as potential egress points for a demand consisting of aggregated voice traffic. Although there will typically be a single preferred gateway for any call, in some cases other gateways could also potentially forward the call. Furthermore, taking a more global perspective, it may be preferable to use one or more of these alternatives to reduce the overall network load. Although this argument sounds plausible in theory, the current traffic matrix generation strategy, [Pollock 2004], does not support multiple egresses. This makes it difficult to assess the utility of such a feature in existing MPLS/VoIP deployments. Multiple egresses could also be potentially useful in the hierarchical demands approach, as described in [Mitchell 2004b], but these ideas would have to be explored further to reach any firm conclusions.

The current prototype does not scale to more than a few tens of nodes. The modelling system in use, MPL/XA, tends to crash or hang when presented with large examples, making it difficult to explore the problem size boundaries in detail. The system is also very sensitive to how the model is encoded in MPL, for example the parameter order in relations. It seems likely that the performance could be improved by using an MPL consultant to tune the model. Some very large multicommodity flow problems have been solved in other contexts. However, small changes to a model can have a large effect on the computational complexity, making it difficult to draw too many conclusions or expectations from such results. It would also be interesting to reimplement the model with other modelling languages and solvers, to assess the extent to which MPL is the bottleneck. Unfortunately, we do not have access to any other solvers for comparison at this time. There has been some preliminary work on the hierarchical decomposition of demands. This approach could allow us to scale to much larger networks, but it depends crucially on the underlying network topology. It is difficult to assess whether such techniques are useful in practice, and/or what further heuristics might need to be deployed, without access to representive topology data.

In this paper we have argued there is a need for both edge and path-based modelling of demands. Each approach has its advantages and disadvantages. By supporting both techniques within a single model we can optimize demands associated with many different traffic classes. For demands with relatively strict QoS constraints our results show that a path-based approach, with around ten paths, produces reasonable results. However, for demands associated with more liberal traffic classes, for example best-effort traffic, the edge-based approach is the preferred solution. The edge-based model may also be useful for suggesting additional candidate paths for a path-based demand. A drawback of the path-based approach is that our solutions are only as good as the set of paths we have chosen. The edge-based approach does not suffer from this deficiency, but may generate solutions that violate the QoS constraints. If we could express end-to-end constraints in the edge model then we could avoid this problem. However, it is not clear to what extent we could do this, and our initial results are not encouraging.

Any optimization strategy that is used in a real deployment is likely to be a compromise. We would like to reduce average link utilization, particularly where we are charged for link usage. However, we don't want to take this goal to such an extreme that some links are used to their full capacity. In such cases we may prefer a slightly worse average utilization if this gives us spare capacity on all our links. We might also wish to reduce the number of paths used, as each one will translate into an LSP, with its associated

state and signalling overheads. We could build such compromise strategies in multiple ways. The hybrid strategy approach, based on the work by [Erbas and Mathar 2002], allows us to build a hybrid out of two or more simpler strategies, together with the weighting we wish to apply to each one. This is a very flexible approach, but has the disadvantage that we need to run the optimizer $n+1$ times, where n is the number of strategies in the hybrid. An alternative approach, as illustrated by the KohlerHybrid strategy, attempts to balance competing goals within a single objective function. Constructing such a function can be challenging, and it may also be difficult to adjust the trade-off weighting, or add additional criteria. However, our results show that the monolithic approach is substantially faster than the structured approach, and can often yield results that are at least as good. It seems likely that the structured approach, whilst elegant in principle, will be too expensive, computationally, to be realistic in practice.

The candidate path selection process works well where the network topology is loosely connected, and/or where there are strict QoS constraints. In these cases the number of acceptable paths may be less than N, the number of paths we are looking for. The approach becomes more problematic when the number of potential paths is very much larger than N. In these cases we may find that computing the first N paths gives us insufficient variety. Adding additional eligible paths from an edge-based solution is one approach that may be worth pursuing, particularly if we store the candidate paths across runs, allowing us to amortize the cost. The hierarchical demands approach also indirectly addresses this problem, as the candidate path problem on a large graph is replaced by multiple problems on smaller graphs. The TOAD currently calculates the candidate paths for all demands prior to running the optimizer. When using priorities it would make more sense to generate the candidate paths at the start of each priority step. This would allow lower priority demands to construct candidate path sets based on the residual capacity of the links.

If the TOAD were to be deployed in a production environment it would need a provisioning system. We could drive such a system from the routes file. The provisioning tool would have to compare the desired configuration with the current state to determine the set of changes that would need to be made. However, if we adopted the transition policy approach described in Section 12.2, then the TOAD itself would "know" the current state, and so it may be more natural for the TOAD to output just the changes. In either case we need to consider precisely what we wish to provision. We could map each path into a strictly-routed LSP, but this would make the solution very fragile. Fast-reroute link protection mechanisms may protect us from link failures, but if a node fails then we may break the tunnel completely. This raises the question of whether it might be preferable to convert the exact routes into partial ones, giving us a more robust solution that may be easier to deploy, whilst providing most of the savings of the offline solution. For example, when routing a demand from Los Angeles to New York, we might determine the best approach is to split the demand into two LSPs, one via Chicago, and the other via Houston. Whilst the solution will compute the exact route to be followed by each LSP, it may be sufficient to provision the LSPs using a loose specification, with a single intermediate hop constraint for each one. By allowing some flexibility in how the traffic reaches Chicago, and from there to New York, we may be able to make the solution more resilient, whilst still achieving the broad traffic engineering goals of the computed solution. Of course the problem now becomes one of identifying which of the intermediate hops in the computed solution should be kept, and which can be dropped. A simple heuristic might be to drop any hop that is on the shortest path from the source to the hop destination. In the common case where an LSP follows the shortest path from source to destination then no intermediate routers would need to be specified. However, where the traffic is routed away from a predicted hotspot then we would be forced to include additional loose hops. The downside of such an approach is that we start to lose any QoS guarantees, particularly when links start to fail. But this may still be the most appropriate compromise, given the cost involved in computing backup paths that can cope with multiple failures, whilst also providing QoS guarantees.

In conclusion, this paper has described the main features of a prototype MPLS traffic engineering system, and an evaluation of its performance. It is difficult to draw too many firm conclusions given the limited access we currently have to representative data for network topologies and traffic demands. If this situation changed then it would be worth revisiting some of this analysis to evaluate how well such a system would perform in practice, and to better quantify any operating efficiency gains. Similarly, if we had access to a provisioning system then it may be worth combining the tools, enabling a more complete exploration and

analysis of the Traffic Engineering life cycle. There are many areas of the TOAD that could benefit from further development, if market conditions changed sufficiently to warrant such effort. Reimplementing the model using other modeling languages and solvers, for example AMPL/CPLEX, would allow us to assess the extent to which MPL forms a bottleneck. Automating the provisioning of the calculated solutions would provide a more complete solution to the TE problem. The provisioning process itself is not particularly difficult, other than the necessity of having to support multiple router vendors. The real challenge comes when we need to incrementally reprovision the system, potentially having to roll back changes in the event of failure. The desire to minimize such changes, whilst ensuring the deployed solutions do not depart too far from the computed optimum, is clearly an area that could benefit from further exploration. Scalability is another area that deserves more attention in the future. Preliminary work has suggested some promising avenues for exploration, but these should be evaluated more rigorously if topology data and marketing requirements ever become available.

A. TEMPLATES

A typical template might start with the following definitions:

```
<template name = "DemandPlacement"
          type = "DemandPlacement"><![CDATA[
  TITLE
    $strategy;

  INDEX
    node := 0..$numVertices - 1;
    src  := node;
    dst  := node;

    trafficClass := 0..$numTrafficClasses - 1;

    demand := 1..$numDemands;
    demandP[demand] := INDEXFILE("Admissible.dat", 1);
    demandE[demand] := demand - demandP
```

The metavariables are of the form $x. So, for example, `$numVertices` will be replaced by the number of vertices in the current problem when the template is expanded into a model. The file `Admissible.dat` is generated by the TOAD and relates each demand to the set of paths that are admissible for the demand. The domain of this map defines the set of path-based demands, called `demandP` in the model. The syntax `demandP[demand]` defines `demandP` to be a subset of `demand`. The edge-based demands, called `demandE` in the model, are all those demands that aren't in `demandP`.[1]

The next part of the template specifies the topology, and paths across this topology.

```
INDEX
  edge[src, dst] := INDEXFILE("Edges.dat", 1);
  path := 0..$numPaths - 1;
  traverses[path, edge] := INDEXFILE("Traverses.dat");
  admissible[demandP, path] := INDEXFILE("Admissible.dat");

DENSE DATA
  bandwidth[edge]     := $bandwidth;
  delay[edge]         := SPARSEFILE("Edges.dat", 3);
  requested[demand]   := SPARSEFILE("Demands.dat", 3);
```

The `edge` index defines the subset of the cross-product between nodes forming the edges of the network. The file `Edges.dat` is generated by the TOAD. The index `traverses` defines which paths traverse each edge.

[1]To work around bugs in MPL, the exact definition of these index sets has to be slightly more complex in reality.

The index `admissible` defines which paths are acceptable for each of the path-based demands. Each edge has a bandwidth and a delay, and there is a bandwidth requirement associated with each demand.

Each demand has an associated traffic class. It also has one or more ingresses and egresses, depending on how flexible we wish the model to be.

```
INDEX
  demandClass[demand].trafficClass   := INDEXFILE("DemandClass.dat");
  demandClassFn[demand,trafficClass] := INDEXFILE("DemandClass.dat");

  ingress[demand, node]  := INDEXFILE("Ingresses.dat");
  egress[demand, dst]    := INDEXFILE("Egresses.dat");
```

This fragment illustrates one of the (many) oddities of MPL. The association between demands and traffic classes can either be viewed as a subset of the crossproduct, or as a total function from demands to traffic classes. The two indexes, `demandClass` and `demandClassFn`, reflect these two views. The reason for defining both of them is that the function view isn't supported by MPL in all contexts. For conciseness and clarity we use the functional index where possible, but have to use `demandClass` in some contexts.

The TOAD allows priorities to be associated with traffic classes. The classes with the highest priority are optimized first. The classes with the next highest priority are then optimized on the residual network. And so on. We could generate a separate model, and data files, for each priority level. An alternative approach, adopted by the TOAD, is to specify the current set of "active" traffic classes. Only those demands whose traffic class is in the active set will be optimized by the model. The following index sets encode this approach

```
INDEX
  active[trafficClass] := $pclass;
  selected [demand]
    WHERE FORSOME(trafficClass IN active: trafficClass IN demandClassFn);
  selectedE[demandE]
    WHERE FORSOME(trafficClass IN active: trafficClass IN demandClassFn);
  selectedP[demandP]
    WHERE FORSOME(trafficClass IN active: trafficClass IN demandClassFn);
```

Given a set of active traffic classes, via the `$pclass` metavariable, `selected` is the set of active demands, `selectedE` the subset of these that are edge based, and `selectedP` the path-based demands. During each run of the model it is the demands in `selectedP` and `selectedE` that are optimized. Having optimized the highest priority demands, the remaining demands must be optimized on the residual network. The `$bandwidth` metavariable takes care of this. Initially it is expanded into an expression representing the bandwidth of each each. For subsequent optimization steps it is replaced by an expression denoting the current residual network.

We allow the model to have multiple objectives. For example, in a hybrid strategy we may wish to optimize a number of different objectives simultaneously. The `objective` index enumerates the names of the different objectives.

```
INDEX
  objective := (
    TotalDeficit,
    $objectives
  );
```

The optimization process computes the values of a set of decision variables, whilst minimizing or maximising some objective function. Some of the decision variables depend on the particular strategy being used, whilst others are common to all strategies.

```
DECISION VARIABLES
  $decisionVariables
```

```
  Value[objective]
    EXPORT Activity TO SparseFile("ObjectiveValues");

  AllocatedE[demandE IN selectedE, edge]
    EXPORT Activity TO SparseFile("AllocatedEdges.dat");

  AllocatedP[demandP IN selectedP, path IN admissible]
    EXPORT Activity TO SparseFile("AllocatedPaths.dat");

  Residual[edge]
    EXPORT Activity TO $residual;

  Allocated[edge];
  Deficit[demand IN selected];
```

Some of these decision variables are written to files at the end of the optimization process. These form the solution to the problem, and are read and visualized by the TOAD after the optimization has completed. The values of the decision variables are subject to various constraints. Some of these constraints are dependent on the particular strategy being used, whilst others encode general constraints that should always hold. The constraints have to deal with both path and edge-based demands.

```
SUBJECT TO
! Common constraints

  EdgeAllocation[edge] :
    Allocated =
      SUM(demandP IN selectedP, path IN traverses: AllocatedP)
    + SUM(demandE IN selectedE: AllocatedE);

  ResidualBandwidth[edge] :
    Residual = bandwidth - Allocated;

  TotalDeficitValue :
    Value[TotalDeficit] = SUM(demand IN selected: Deficit);

! Edge demand constraints

  IngressFlowsIn[demandE IN selectedE, node IN ingress] :
    SUM(edge OVER src: AllocatedE[demandE, edge]) = 0.0;

  IngressFlowsOut[demandE IN selectedE, node IN ingress]  :
    SUM(edge OVER dst: AllocatedE[demandE, edge])
    = requested - Deficit;

  TransitBalance[demandE IN selectedE, node]
      WHERE NOT (node IN ingress) AND NOT (node IN egress) :
    SUM(edge OVER src: AllocatedE[demandE, edge]) =
    SUM(edge OVER dst: AllocatedE[demandE, edge]);

  EgressFlowsIn[demandE IN selectedE] :
    SUM(node IN egress,
        edge OVER src: AllocatedE[demandE, edge])
    = requested - Deficit;

  EgressFlowsOut[demandE IN selectedE, node IN egress] :
    SUM(edge OVER dst: AllocatedE[demandE, edge]) = 0.0;
```

```
! Path demand constraints

  SatisfyDemand[demandP IN selectedP] :
    SUM(path IN admissible : AllocatedP) = requested - Deficit;

  $constraints
```

These constraints enforce the preservation of flow, and ensure that the flow over a link is never more than the capacity of the link. The `Deficit` variables record how much of each demand cannot be satisfied in the solution. The objective functions should heavily penalize non-zero values for these variables. We ensure this by defining the objective to be

```
MIN $objectiveFunction
  + 10000000 Value[TotalDeficit];
```

where `$objectiveFunction` is expanded into the objective for the strategy in use.

REFERENCES

Aboul-Magd, O., Andersson, L., and Ashwood-Smith, P. 2001. Constraint-based LSP setup using LDP.

Ahuja, R., Magnanti, T., and Orlin, J. 1993. *Network Flows: Theory, Algorithm, and Applications*. Prentice Hall.

Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., and Swallow, G. 2001. RSVP-TE: Extensions to RSVP for LSP tunnels.

Bejerano, Y., Breitbart, Y., Orda, A., Rastogi, R., and Sprintson, A. 2003. Algorithms for computing QoS paths with restoration. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 2. IEEE, 1435 – 1445.

Boutremans, C., Iannaccone, G., and Diot, C. Impact of link failures on VoIP performance.

Cech, S. 2002. A route server for constraint-based routing in MPLS. M.S. thesis, Universität Klagenfurt, Austria.

Cisco. 2002. Congestion avoidance overview. www.cisco.com.

CPlane. 2003. Traffic optimizer product overview. White paper.

Cui, Y., Xu, K., and Wu, J. 2003. Precomputation for multi-constrained QoS routing in high-speed networks. In *INFOCOM 2003*.

Davie, B. and Rekhter, Y. 2000. *MPLS: Multiprotocol Label Switching Technology and Applications*. Morgan Kaufmann.

Durham, D., Boyle, J., Cohen, R., Herzog, S., Rajan, R., and Sastry, A. 2000. The COPS (Common Open Policy Service) protocol. http://www.ietf.org/rfc/rfc2748.txt.

Erbas, S. C. and Mathar, R. 2002. An off-line traffic engineering model for MPLS networks. In *27th Annual IEEE Conference on Local Computer Networks (LCN'02)*. IEEE.

Feldmann, A., Greenberg, A., Lund, C., Reingold, N., and Rexford, J. 1999. Netscope: Traffic engineering for IP networks.

Feldmann, A., Greenberg, A. G., Lund, C., Reingold, N., Rexford, J., and True, F. 2000. Deriving traffic demands for operational IP networks: methodology and experience. In *SIGCOMM*. 257–270.

Grover, W. and Doucette, J. 2001. On the design of span- and path- restorable mesh networks. Tech. rep., TRLabs, University of Alberta. November.

Iovanna, P., Sabella, R., and Settembre, M. 2003. A traffic engineering system for multilayer networks based on the GMPLS paradigm. *IEEE Network*, 28–37.

Katz, D., Kompella, K., and Yeung, D. 2003. Traffic engineering (TE) extensions to OSPF version 2. IETF RFC 3630.

Köhler, S. and Binzenhöfer, A. 2003. MPLS traffic engineering in OSPF networks - a combined approach. Tech. Rep. 304, Institute of Computer Science, University of Würzburg. February.

Lee, Y., Seok, Y., and Choi, Y. 2004. Traffic engineering with constrained multipath routing in MPLS networks. *IEICE Transactions on Communications E87B,* 5 (May).

Lehane, A. 2002. The IGP Detective. Tech. Rep. AGL-2002-9, Agilent Labs.

Liu, G. and Ramakrishnan, K. G. 2001. A*prune: An algorithm for finding k shortest paths subject to multiple constraints. In *INFOCOM*. 743–749.

Lugrin, J.-M. 2004. Fesi: A free EcmaScript interpreter. http://www.lugrin.ch/fesi/.

Maximal. 2004. The MPL modeling system. http://www.maximal-usa.com/mpl/.

Menth, M. and Hauck, N. 2001. A graph theoretical concept for LSP hierarchies. Tech. Rep. 287, University of Würzburg, Institute of Computer Science. November.

Mitchell, K. 2004a. Bidirectional demands with multiple egresses. Tech. Rep. TOAD working paper, Agilent Labs.

Mitchell, K. 2004b. Hierarchical demands. Tech. Rep. AGL-2004-5, Agilent Labs.

Mitra, D. and Ramakrishnan, K. 2001. Techniques for traffic engineering of multiservice, multipriority networks. *Bell Labs Technical Journal 6,* 1 (January), 139–151.

OPNET. 2003. SP Guru. http://www.opnet.com/products/spguru/plan.html.

Orda, A. and Sprintson, A. 2000. QoS routing: The precomputation perspective. In *INFOCOM (1)*. 128–136.

Pollock, G. 2004. VoIP traffic matrix generation. Tech. Rep. In preparation, Agilent Labs.

RAD. 2004. RAD introduces TDMoIP PMC for OEM developers. http://www.rad.com/Article/0,6583,18087,00.html.

Rosen, E., Viswanathan, A., and Callon, R. 2001. Multiprotocol label switching architecture. http://www.ietf.org/rfc/rfc3031.txt.

Sidhu, D., Nair, R., and Abdallah, S. 1991. Finding disjoint paths in networks. In *SIGCOMM*. 43–51.

Suri, S., Waldvogel, M., and Warkhede, P. R. 2001. Profile-based routing: A new framework for MPLS traffic engineering. In *Quality of future Internet Services*, F. Boavida, Ed. Number 2156 in Lecture Notes in Computer Science. Berlin, 138–157. An earlier version is available as Washington University Computer Science Technical Report WUCS-00-21, July 2000.

Wandl. 2002. IP/MPLSView: Integrated network planning, configuration management & performance management. White paper.

Wang, J., Patek, S., Wang, H., and Liebeherr, J. 2002. Traffic engineering with AIMD in MPLS networks. In *Protocols for High-Speed Networks*. 192–210.